

DENOTACYJNA INŻYNIERIA JĘZYKÓW PROGRAMOWANIA

by twórca systemu mógł wziąć za niego odpowiedzialność,
a podręcznik użytkownika był kompletny i zrozumiały

Książka in statu nascendi

Andrzej Jacek Blikle
przy współpracy z Piotrem Chrzastowskim-Wachtlem

*To zawsze wydaje się niemożliwe,
póki się tego nie zrobi*

Nelson Mandela

Warszawa, 6 marca 2019 r.

© **Copyright by Andrzej Blikle**. W ramach moich praw autorskich chronionych ustawą Prawo autorskie i prawa pokrewne z dnia 4 lutego 1994 (z późniejszymi zmianami) wyrażam zgodę na niekomercyjne rozpowszechnianie niniejszego materiału przez jego zwielokrotnianie bez ograniczeń co do liczby egzemplarzy (w formie elektronicznej lub drukowanej), a także umieszczanie go na stronach internetowych, jednakże bez dokonywania jakichkolwiek zmian i skrótów w tekście i bez zmiany elektronicznego formatu „pdf” na inny. Wszelkie inne rozpowszechnianie niniejszego materiału, w tym jego części, wymaga mojej zgody wyrażonej na piśmie. Dozwolone jest natomiast cytowanie materiału zgodnie z zasadami ustanowionym przez ww. ustawę.



„Denotacyjna inżynieria języków programowania” by Andrzej Blikle in cooperation with Piotr Chrzastowski-Wachtel is licensed under a Creative Commons: Uznanie autorstwa — Użycie niekomercyjne — Bez utworów zależnych 3.0 Polska, szczegóły na <http://creativecommons.org/licenses/by-nc-nd/3.0/pl/>

O aktualnych wersjach książki

Istniejące obecnie wersje językowe książki — polska i angielska — są na bieżąco korygowane i modyfikowane dzięki licznym uwagom, które otrzymuje od moich czytelników. Od grudnia 2018 r. obie są dostępne do pobrania w wersji PDF z mojej witryny

<http://www.moznainaczej.com.pl/inzynieria-denotacyjna> — wersja polska

<http://www.moznainaczej.com.pl/denotational-engineering> — wersja angielska

Książka jest również dostępna na moim koncie ResearchGate.

Wszystkich czytelników gorąco zachęcam do nadsyłania swoich uwag zarówno na tematy merytoryczne, jak też redakcyjne i językowe. Za wszystkie będę niezmiernie wdzięczny. Piszcie do mnie na adres: andrzej.blikle@moznainaczej.com.pl.

Podziękowania do wersji polskiej

W czerwcu 2018 r. książka w wersji roboczej została udostępniona wybranym czytelnikom, a wraz z tym wydarzeniem zacząłem otrzymywać uwagi na jej temat. Poniżej lista osób, których uwagi doprowadziły do zmian i ulepszeń książki. Nazwiska podaję w kolejności chronologicznej, w jakiej otrzymywałem uwagi. Ta lista — mam nadzieję — będzie się powiększać. Wszystkim za wkład w moją książkę gorąco dziękuję.

Piotr Chrzastowski-Wachtel, Stanisław Budkowski, Antoni Mazurkiewicz, Marek Ryćko, Bogusław Jackowski, Ryszard Kubiak, Paweł Urzyczyn, Stefan Sokołowski, Marek Bednarczyk, Wiesław Pawłowski, Jan Madey, Krzysztof Apt, Andrzej Tarlecki, Jarosław Deminet.

Uwaga techniczna dotycząca formatowania

Jeżeli niniejszy plik jest otwarty w systemie Word (bo istnieje wersja pdf), to aby formuły się nie rozjechały, należy ustawić tabulator na 0,5 cm. Jest to lokalny atrybut dokumentu. Ustawiamy go w panelu narzędziowym w grupie

Narzędzia główne / Akapit

gdzie należy kliknąć małą strzałkę znajdującą się w prawym dolnym narożniku sekcji.

Cytat z Nelsona Mandeli za: Steve Case „Trzecia fala. Wizja gospodarki przyszłości”, Studio Emka 2017, str. 238

Spis treści

PRZEDMOWA	8
1 WPROWADZENIE.....	12
1.1 ODWRÓĆMY TRADYCYJNĄ KOLEJ RZECZY	12
1.2 CO ZAWIERA KSIĄŻKA	15
1.3 CZEGO KSIĄŻKA NIE OFERUJE	17
1.4 LINGUA Z LOTU PTAKA	17
1.4.1 Dane i ich typy	18
1.4.2 Błędy abstrakcyjne	21
1.4.3 Wyrażenia.....	22
1.4.4 Instrukcje.....	23
1.4.5 Deklaracje zmiennych i definicje typów	25
1.4.6 Procedury.....	26
1.4.7 Programowanie obiektowe.....	27
1.4.8 Programowanie SQL-owe	27
1.4.9 Programy.....	28
1.4.10 Programowanie walidujące.....	28
2 METASOFT I JEGO MATEMATYKA	30
2.1 PODSTAWOWE KONWENCJE NOTACYJNE METASOFT.....	30
2.1.1 Notacja ogólnie-matematyczna	31
2.1.2 Zbiory.....	31
2.1.3 Funkcje.....	32
2.1.4 Krotki	36
2.2 ZBIORY CZĘŚCIOWO UPORZĄDKOWANE.....	38
2.3 ZBIORY ŁAŃCUCHOWO ZUPEŁNE	39
2.4 ZŁŻ JĘZYKÓW FORMALNYCH	42
2.5 GRAMATYKI RÓWNANIOWE	44
2.6 ZŁŻ RELACJI BINARNYCH.....	46
2.7 ZŁŻ DZIEDZIN DENOTACYJNYCH.....	49
2.8 BŁĘDY ABSTRAKCYJNE.....	51
2.9 TRÓJWARTOŚCIOWY RACHUNEK ZDAŃ.....	52
2.10 ALGEBRY DANYCH	55
2.11 ALGEBRY WIELORODZAJOWE	58
2.12 SKŁADNIA ABSTRAKCYJNA I ALGEBRY OSIĄGALNE	62
2.13 ALGEBRY JEDNOZNACZNE I WIELOZNACZNE	66
2.14 ALGEBRY I GRAMATYKI.....	68
3 SEMANTYCZNA POPRAWNOŚĆ PROGRAMÓW	76
3.1 UWAGI HISTORYCZNE.....	76
3.2 PROGRAMY ITERACYJNE.....	78
3.3 PROCEDURY I REKURENCJA	80
3.4 DWA POJĘCIA POPRAWNOŚCI PROGRAMÓW	82
3.5 POPRAWNOŚĆ CZĘŚCIOWA.....	86
3.5.1 Złożenie i rozgałęzienie	86
3.5.2 Rekurencja i iteracja	88
3.6 POPRAWNOŚĆ CAŁKOWITA	92
3.6.1 Złożenie i rozgałęzienie	92
3.6.2 Rekurencja i iteracja	94
4 O MODELACH DENOTACYJNYCH OGÓLNIE.....	99
4.1 JAK DO TEGO DOSZŁO?	99
4.2 OD DENOTACJI DO SKŁADNI.....	103

4.3	JĘZYKI Z SERII LINGUA	103
4.4	PO CO NAM DENOTACYJNE MODELE JĘZYKÓW PROGRAMOWANIA?	104
4.5	PIĘĆ KROKÓW DO MODELU DENOTACYJNEGO.....	105
4.6	METAKONWENCJE NOTACYJNE I JĘZYKOWE	108
5	ALGEBRAICZNO-DENOTACYJNY MODEL STRUKTUR DANYCH	110
5.1	OGÓLNA IDEA MODELU	110
5.2	ALGEBRY STRUKTUR DANYCH.....	112
5.2.1	<i>Algebra danych</i>	112
5.2.2	<i>Algebra korpusów</i>	117
5.2.3	<i>Algebra kompozytów</i>	123
5.2.4	<i>Algebra transferów</i>	127
5.2.5	<i>Algebra typów</i>	134
5.3	ALGEBRA DENOTACJI WYRAŻEŃ.....	138
5.3.1	<i>Wartości i stany pamięci</i>	138
5.3.2	<i>Denotacje wyrażeń danologicznych</i>	141
5.3.3	<i>Postać wyraźna definicji konstruktorów denotacji</i>	144
5.3.4	<i>Denotacje wyrażeń typologicznych</i>	145
5.3.5	<i>Algebra denotacji wyrażeń danologicznych, typologicznych i transferowych</i>	146
5.3.6	<i>Sześć kroków do algebry denotacji wyrażeń</i>	147
5.4	ALGEBRY SKŁADNI.....	148
5.4.1	<i>Składnia abstrakcyjna języka Lingua-A</i>	148
5.4.2	<i>Składnia konkretna języka Lingua-A</i>	152
5.4.3	<i>Składnia kolokwialna języka Lingua-A</i>	156
5.4.3.1	Reguły uniwersalne	156
5.4.3.2	Danologiczne wyrażenia boolowskie.....	156
5.4.3.3	Danologiczne wyrażenia liczbowych.....	157
5.4.3.4	Danologiczne wyrażenia tablicowe	157
5.4.3.5	Danologiczne wyrażenia rekordowe	158
5.4.3.6	Transferowe wyrażenia tablicowe	159
5.4.3.7	Typologiczne wyrażenia rekordowe	159
5.4.3.8	Transferowe wyrażenia rekordowe	159
5.4.3.9	Wyrażenia typologiczne	160
5.5	ZADANIA BUDOWNICZEGO JĘZYKA.....	161
5.6	DWIE FORMUŁY PODRĘCZNIKA JĘZYKA PROGRAMOWANIA	162
5.7	SZKIC SEMANTYKI LINGUA-A	163
6	LINGUA-1 — JĘZYK IMPERATYWNY BEZ PROCEDUR.....	167
6.1	DENOTACJE	167
6.1.1	<i>Dziedziny denotacyjne</i>	167
6.1.2	<i>Deklaracje zmiennych danologicznych</i>	168
6.1.3	<i>Definicje stałych typologicznych</i>	169
6.1.4	<i>Instrukcja przypisania</i>	169
6.1.5	<i>Instrukcja wymiany transferu</i>	172
6.1.6	<i>Instrukcja trywialna</i>	172
6.1.7	<i>Instrukcje strukturalne</i>	173
6.1.8	<i>Obsługa błędów</i>	174
6.1.9	<i>Preambuły i programy</i>	175
6.1.10	<i>Podsumowanie dotyczące roli typów w programach</i>	176
6.2	SKŁADNIA	177
6.2.1	<i>Składnia abstrakcyjna</i>	177
6.2.2	<i>Składnia konkretna</i>	177
6.2.3	<i>Składnia kolokwialna</i>	182
6.2.4	<i>Przykład prostego programu</i>	182
6.3	SEMANTYKA.....	184
7	LINGUA-2 — PROCEDURY	186
7.1	WPROWADZENIE DO MODELU DLA PROCEDUR.....	186
7.1.1	<i>O procedurach historycznie</i>	186
7.1.2	<i>Procedury a programowanie strukturalne</i>	187
7.1.3	<i>O procedurach denotacyjnie</i>	188

7.1.4	<i>Dziedziny denotacyjne związane z procedurami</i>	190
7.2	KOMUNIKACJA PROCEDURY IMPERATYWNEJ Z PROGRAMEM.....	191
7.2.1	<i>Jak to działa?</i>	192
7.2.2	<i>Zgodność list parametrów</i>	194
7.2.3	<i>Przekazywanie parametrów aktualnych do procedury</i>	196
7.2.4	<i>Zwracanie parametrów referencyjnych do programu</i>	197
7.3	PROCEDURY IMPERATYWNE Z REKURENCJĄ POJEDYNCZĄ.....	199
7.3.1	<i>Konstruktor parametrów</i>	199
7.3.2	<i>Konstruktor procedury</i>	200
7.3.3	<i>Instrukcja wywołania procedury imperatywnej</i>	202
7.3.4	<i>Deklaracja procedury imperatywnej</i>	203
7.4	PROCEDURY IMPERATYWNE Z REKURENCJĄ WZAJEMNĄ	204
7.4.1	<i>Rekurencja wzajemna</i>	204
7.4.2	<i>Konstruktor wieloprocedury</i>	205
7.4.3	<i>Instrukcja wywołania składowej wieloprocedury</i>	206
7.4.4	<i>Deklaracja wieloprocedury</i>	206
7.5	PROCEDURY FUNKCYJNE	206
7.5.1	<i>Struktura deklaracji procedury funkcyjnej</i>	207
7.5.2	<i>Dziedziny procedur funkcyjnych</i>	208
7.5.3	<i>Konstruktor procedury funkcyjnej</i>	208
7.5.4	<i>Wyrażenie wywołania procedury funkcyjnej</i>	210
7.5.5	<i>Deklaracja procedury funkcyjnej</i>	211
7.6	PROCEDURY JAKO PARAMETRY PROCEDUR	211
7.7	PROGRAMY	212
7.8	SKŁADNIA I SEMANTYKA	212
7.8.1	<i>Sygnatura algebry denotacji</i>	212
7.8.1.1	<i>Nośniki algebry denotacji</i>	213
7.8.1.2	<i>Konstruktory algebry denotacji</i>	213
7.8.2	<i>Składnia konkretna</i>	215
7.8.3	<i>Składnia kolokwialna</i>	219
7.8.4	<i>Semantyka</i>	219
7.8.4.1	<i>Parametry aktualne</i>	219
7.8.4.2	<i>Parametry formalne</i>	219
7.8.4.3	<i>Wyrażenia danologiczne: wywołanie procedury funkcyjnej</i>	219
7.8.4.4	<i>Instrukcje: wywołanie procedury imperatywnej</i>	219
7.8.4.5	<i>Deklaracja procedury imperatywnej</i>	220
7.8.4.6	<i>Deklaracja wieloprocedury imperatywnej</i>	220
7.8.4.7	<i>Deklaracje procedury funkcyjnej</i>	220
7.8.4.8	<i>Preambuły</i>	220
7.8.4.9	<i>Programy</i>	220
8	LINGUA-2V — PROGRAMOWANIE WALIDUJĄCE	222
8.1	STRUKTURA JĘZYKA	222
8.2	WARUNKI	223
8.2.1	<i>O warunkach ogólnie</i>	224
8.2.2	<i>Warunki danologiczne</i>	226
8.2.3	<i>Warunki walidacyjne</i>	226
8.2.4	<i>Warunki algorytmiczne</i>	229
8.3	INSTRUKCJE SPECYFIKOWANE	229
8.4	TEZY	233
8.4.1	<i>Właściwości składniowe</i>	233
8.4.2	<i>Metawarunki</i>	234
8.4.3	<i>Metaprogramy</i>	238
8.4.4	<i>Paradoks de Bakera w logice Hoare'a</i>	240
8.5	KONSTRUOWANIE METAPROGRAMÓW POPRAWNYCH	241
8.5.1	<i>Konwencja notacyjna</i>	241
8.5.2	<i>Reguły podstawowe</i>	242
8.5.3	<i>Wywołanie procedury imperatywnej</i>	245
8.5.4	<i>Przypadek procedury rekurencyjnej</i>	247
8.5.5	<i>Wywołanie procedury funkcyjnej</i>	248
8.6	PROGRAMOWANIE TRANSFORMACYJNE	249

8.6.1	<i>Pierwszy przykład</i>	249
8.6.2	<i>Dodawanie identyfikatora rejestrowego</i>	257
8.6.3	<i>Zmiana typu danych</i>	260
8.7	NIEZMIENNIKI VERSUS ASERCJE	263
9	LINGUA-3 — PROGRAMOWANIE OBIEKTOWE	265
9.1	ZAŁOŻENIA MODELU	265
9.2	WYRAŻENIA OBIEKTOWE.....	266
9.3	DEKLARACJE OBIEKTÓW	267
9.4	PRZYWOŁYWANIE OBIEKTÓW W PROGRAMACH.....	268
9.5	PREFIKSOWANIE PROGRAMÓW PRZYWOŁANIAMI OBIEKTÓW.....	269
9.6	ROZSZERZENIE ALGEBRY SKŁADNI.....	269
9.7	PROGRAMOWANIE WALIDACYJNE W LINGUA-3	271
10	OBIEKTY ZEWNĘTRZNE — ZARYS IDEI	272
11	RELACYJNE BAZY DANYCH INTUICYJNIE	274
11.1	UWAGI WSTĘPNE	274
11.2	DANE PROSTE	275
11.3	TWORZENIE TABEL	278
11.4	RELACJE NADRZĘDNOŚCI TABEL	281
11.5	INSTRUKCJE MODYFIKOWANIA TABEL	283
11.6	TRANSAKCJE.....	284
11.7	ZAPYTANIA.....	286
11.8	FUNKCJE AGREGUJĄCE	289
11.9	PERSPEKTYWY	289
11.10	KURSORY.....	291
11.11	ŚRODOWISKO KLIENT-SERWER	291
12	LINGUA-SQL	293
12.1	OGÓLNE ZAŁOŻENIA DOTYCZĄCE MODELU.....	293
12.2	KOMPOZYTY	293
12.2.1	<i>Dane, korpusy i kompozyty SQL-owe</i>	293
12.2.2	<i>Relacje podrzędności tabel</i>	296
12.2.3	<i>Sygnatura nowych konstruktorów algebry kompozytów</i>	297
12.2.4	<i>Konstruktory kompozytów prostych</i>	299
12.2.5	<i>Konstruktory kompozytów wierszowych</i>	299
12.2.6	<i>Wierszowe konstruktory kompozytów tabelowych</i>	301
12.2.7	<i>Kolumnowe konstruktory kompozytów tabelowych</i>	306
12.2.8	<i>Referencyjny konstruktor kompozytów tabelowych</i>	310
12.3	KORPUSY	313
12.4	TRANSFERY	314
12.4.1	<i>Transfery wierszowe</i>	314
12.4.2	<i>Transfery tabelowe</i>	315
12.5	TYPY.....	316
12.6	WARTOŚCI BAZODANOWE.....	318
12.7	ALGEBRA DENOTACJI.....	319
12.7.1	<i>Stany i dziedziny denotacyjne</i>	319
12.7.2	<i>Denotacje wyrażeń danologicznych</i>	320
12.7.3	<i>Denotacje wyrażeń typologicznych i transferowych</i>	322
12.7.4	<i>Denotacje definicji stałych typologicznych</i>	323
12.7.5	<i>Denotacje deklaracji zmiennych danologicznych</i>	324
12.7.6	<i>Instrukcje</i>	325
12.7.6.1	<i>Kategorie instrukcji SQL-owych</i>	325
12.7.6.2	<i>Instrukcje wierszowe</i>	325
12.7.6.3	<i>Dwa uniwersalne konstruktory przypisań tabelowych</i>	326
12.7.6.4	<i>Transakcje</i>	328
12.7.6.5	<i>Tabelowe instrukcje globalne</i>	333
12.7.6.6	<i>Tabelowe instrukcje lokalne</i>	334
12.7.6.7	<i>Zapytania</i>	335
12.7.6.8	<i>Instrukcja wymiany transferu</i>	335
12.7.6.9	<i>Kursory</i>	335

12.7.6.10	Perspektywy	335
12.7.6.11	Instrukcje bazodanowe	335
12.8	SKŁADNIA KONKRETNA	338
12.9	SKŁADNIA KOŁOKWIALNA	342
12.10	REGUŁY KONSTRUOWANIA PROGRAMÓW POPRAWNYCH	343
13	CO ZOSTAŁO JESZCZE DO ZROBIENIA	344
13.1	PODSTAWY	344
13.1.1	<i>Rozbudowanie modelu dla Lingua</i>	344
13.1.2	<i>Uzupełnienie modelu dla Lingua</i>	344
13.1.3	<i>Zasady pisania podręczników użytkownika</i>	344
13.2	IMPLEMENTACJE	345
13.2.1	<i>Narzędzia dla projektantów języków</i>	345
13.2.2	<i>Narzędzia dla programistów w Lingua</i>	345
13.2.3	<i>Podręczniki</i>	345
13.3	EKSPERYMENTY PROGRAMISTYCZNE	345
13.4	BUDOWANIA SPOŁECZNOŚCI ZWOLENNIKÓW LINGUA	346
14	ANEKS 1 — DRZEWA UOGÓLNIONE	347
14.1	NOŚNIKI WYJŚCIOWE ALGEBRY DANYCH	347
14.2	KONSTRUKTORY ALGEBRY DANYCH	349
14.3	ALGEBRA KORPUSÓW	350
14.4	ALGEBRA KOMPOZYTÓW	352
14.5	ALGEBRA TRANSFERÓW	353
14.6	ALGEBRA TYPÓW	353
14.7	ALGEBRA DENOTACJI WYRAŻEŃ DANOLOGICZNYCH	353
14.8	ALGEBRA DENOTACJI WYRAŻEŃ TYPOLOGICZNYCH	353
14.9	ALGEBRA DENOTACJI JĘZYKA LINGUA-2D	353
14.10	PROGRAMOWANIE WALIDUJĄCE W LINGUA-2DV	353
15	ANEKS 2 — O PODRĘCZNIKACH UŻYTKOWNIKA	354
15.1	O POTRZEBIE NAUCZANIA INFORMATYKI	354
15.2	GENIUSZE I IDIOCI	355
16	BIBLIOGRAFIA	358
17	SKOROWIDZE	361
17.1	SKOROWIDZ POJĘĆ I NAZWISK	361
17.2	SKOROWIDZ OZNACZEŃ	365
17.3	SKOROWIDZ DZIEDZIN I ALGEBR	365

Przedmowa

Kiedy w roku 1990 postanowiłem „na jakiś czas” zająć się moją rodzinną firmą cukierniczą A. Blikle — co zajęło mi dwie dekady 1990-2010 — miałem już wstępnie przygotowany materiał do książki na temat matematycznych modeli języków programowania. Był on wynikiem badań trwających blisko trzydzieści lat, które rozpocząłem zaraz po ukończeniu studiów na Wydziale Matematyki i Fizyki Uniwersytetu Warszawskiego w roku 1962. Podjąłem wtedy pracę w grupie młodych matematyków stawiających sobie za cel zbudowanie narzędzi matematycznych dla inżynierii oprogramowania. Takich zespołów było wówczas w Polsce kilka, a na całym świecie pewnie 20-30. Choć narzędzia dla inżynierii oprogramowania budowaliśmy w różny sposób, to nasza ocena sytuacji ówczesnej informatyki była w zasadzie jednakowa. Postaram się ją streścić poniżej.

W każdej inżynierii, poza inżynierią oprogramowania, projektowanie nowego produktu rozpoczyna się od stworzenia opisu tego produktu w specyficznym dla danej inżynierii języku, który posługuje się z jednej strony wyspecjalizowanym aparatem pojęciowym, a z drugiej — aparatem matematycznym. Do tego dochodzi nierzadko rysunek techniczny. W tych trzech językach — branżowym, matematycznym i graficznym — powstaje opis przyszłego produktu uzupełniony o komentarze pisane w języku potocznym. Matematyczna część tego opisu, zwana zwykle „obliczeniami”, stanowi gwarancję, że wytworzony na jej podstawie produkt będzie miał oczekiwane przez jego twórców właściwości: most się nie zawali, skrzydło samolotu nie pęknie, instalacja chemiczna będzie prowadzić prawidłową syntezę. Oczywiście zdarzają się awarie mostów, samolotów i instalacji chemicznych, te jednak są albo wynikiem błędów w obliczeniach (a więc w projekcie), albo błędów w wykonaniu produktu, albo wreszcie błędów popełnianych przez użytkownika. Aby uniknąć takich sytuacji, przed oddaniem produktu do eksploatacji prowadzi się „testy eksploatacyjne” pozwalające na wykrycie ewentualnych zagrożeń. Później produkt jest oddawany do użytku i zwykle zachowuje się zgodnie z oczekiwaniami jego twórców.

W inżynierii oprogramowania sytuacja była zgoła inna. Jedyne projekt przyszłego produktu, jaki powstawał przed przystąpieniem do pisania kodu, był mniej lub bardziej pełnym opisem oczekiwanych funkcjonalności produktu stworzonym w języku potocznym. Oczywiście zdarzały się sytuacje, w których częścią opisu był wyrażony w języku matematycznym algorytm, te jednak przypadki zwykle ograniczały się do programów z obszarów matematyki stosowanej, służących do rozwiązywania równań, statystycznej obróbki danych, czy też procedur sortowania. Jednakże w przypadku innych zastosowań informatyki, czy to „szytych na miarę” pod wykonawcę, jak system obsługi ruchu samolotów, czy też tworzonych „na półkę”, jak systemy finansowo-księgowe, przyszły produkt był w całości opisywany w języku potocznym i to najczęściej pozbawionym profesjonalnego aparatu pojęciowego. Wynikiem tej sytuacji był dobrze znany informatykom fakt, że nierzadko więcej niż połowa budżetu przeznaczonego na tworzenie produktu programistycznego pochłaniało tropienie i usuwanie błędów wprowadzonych do systemu na etapie pisania kodu. Warto dodać, że kwota ta obejmowała jedynie koszty ponoszone przed przekazaniem produktu do eksploatacji. Pozostałe ponosił już użytkownik płacąc

za „pielęgnowanie” systemu (ang. „maintenance”), co zresztą do tej pory stanowi poważną pozycję w przychodach firm informatycznych. Znane też były przykłady spektakularnych katastrof, do jakich doprowadziły błędy w systemach oprogramowania:

- Śmierć kilku osób w Ameryce Północnej przy terapii nowotworowej urządzeniem Therac-25 (1985).
- Katastrofa amerykańskiego lądownika na Wenus (lata 80-te).
- Katastrofa platformy wiertniczej w norweskim fiordzie (1991).
- Katastrofa Airbusa w Warszawie (1993)¹.
- Przeoczenie przez niemieckie służby meteorologiczne zbliżania się huraganu Lothar (1999).
- Błędy zaokrągleń w arytmometrze Intelu (rok 1995).

Tak było przedwczoraj i wczoraj. A jak jest dziś? Dziś systemy oprogramowania są nieporównanie większe i bardziej złożone niż dawniej, powstaje ich o kilka rzędów wielkości więcej, a liczba użytkowników informatyki rośnie wykładniczo. Jednakże główne problemy, o których pisałem wyżej w zasadzie się nie zmieniły. Świadczy o tym choćby następująca statystyka dotycząca projektów programistycznych o łącznej wartości 250 mld USD (źródło [1]):

- 88% projektów przekroczyło planowany czas realizacji lub budżet,
- średnie przekroczenie planowanych kosztów wyniosło 189%,
- średnie przekroczenie czasu realizacji wyniosło 222%.

Z krajowego podwórka informatycy, i nie tylko, pamiętają zapewne przygody związane z tworzeniem systemu oprogramowania dla ZUS.

Tak jest w wielkiej informatyce przemysłowej. A jak jest w tej małej, wykorzystywanej dziś już nie przez miliony, ale przez miliardy użytkowników? Oto cytat z umowy licencyjnej dotyczącej pewnego systemu finanse-kadry:

Producent oprogramowania zwraca uwagę, że w oparciu o bieżący stan techniki, nie jest możliwe wyprodukowanie programu komputerowego w taki sposób, aby bezbłędnie pracował we wszystkich możliwych konfiguracjach. Producent gwarantuje w oparciu o dotychczasowych użytkowników, że do dnia zawarcia niniejszej umowy nie zna żadnych błędów w przekazywanym programowaniu.

Zwróćmy uwagę na kuriozalność tej deklaracji: producent gwarantuje, że *nie zna żadnych błędów w przekazywanym programowaniu*. Jedyne więc, co jest w stanie zagwarantować użytkownikowi, który wydaje na system 200 tys. zł, to że odkrywane w przyszłości błędy będą również dla niego prawdziwą niespodzianką. Nie dodał już, że to właśnie pozwoli mu wypuszczać na rynek kolejne „ulepszone” wersje swojego produktu, co będzie objęte dodatkowo płatnym abonamentem serwisowym.

¹ W tym przypadku, choć przyczyna wypadku miała swoje źródło w oprogramowaniu, to jednak błąd zrobili nie programiści, ale konstruktorzy hamulców, którzy nie przewidzieli pewnych szczególnych warunków aerodynamicznych mogących wystąpić przy lądowaniu samolotu. To przeoczenie spowodowało, że przekazali programistom nieprawidłową specyfikację przyszłego programu. Choć więc katastrofę spowodowało oprogramowanie, nie była to wina programistów. Za tę uwagę jestem wdzięczny Jarkowi Deminetowi.

Jest też powszechnie znanym faktem, że użytkownik nabywający aplikację informatyczną musi zaakceptować tzw. „wyłączenie odpowiedzialności” (ang. disclaimer) ze strony producenta aplikacji. A oto typowy przykład takiego wyłączenia (nazwa firmy została zastąpiona słowem „Firma”):

O ile nie zaznaczono inaczej w „Warunkach dodatkowych”, Usługi i Oprogramowanie są udostępniane w stanie, w jakim się znajdują („AS-IS”). W maksymalnym zakresie dozwolonym przez prawo, Firma wyłącza wszelkie gwarancje wyraźne lub domniemane, w tym domniemane gwarancje nienaruszania praw, przydatności handlowej i przydatności do określonego celu. Firma nie przyjmuje żadnych zobowiązań dotyczących treści zawartej w Usługach. Ponadto Firma nie gwarantuje, że:

- a) Usługi lub Oprogramowanie spełnią wymagania Użytkownika, będą stale dostępne oraz że będą działały w sposób nieprzerwany, terminowy i bezbłędny;*
- b) efekty uzyskane w wyniku użycia Usług lub Oprogramowania będą skuteczne, dokładne i niezawodne;*
- c) jakość Usług i Oprogramowania spełni oczekiwania Użytkownika; oraz*
- d) błędy lub usterki w Usługach lub Oprogramowaniu zostaną naprawione.*

To wyłączenie pochodzi z 2018 roku i dotyczy nie małej lokalnej firmy, ale dużej i międzynarodowej.

Czy ktoś mógłby sobie wyobrazić, że producent jakiegokolwiek nieinformatycznego produktu przemysłowego — samochodu, pralki, telewizora, czy budynku — mógłby zażądać od swoich klientów zgody na podobne zrzeczenie się swoich praw? Dlaczego więc w przemyśle IT jest to zjawisko powszechne?

Myślę, że odpowiedzi na to pytanie udzielił wspomniany wyżej producent systemu finansowo-księgowego: *w oparciu o bieżący stan techniki, nie jest możliwe wyprodukowanie programu komputerowego w taki sposób, aby bezbłędnie pracował we wszystkich możliwych konfiguracjach*. Prawdę mówiąc całkowicie podzielam ten pogląd. Żadna firma z branży IT nie jest w stanie wziąć za swój produkt takiej odpowiedzialności, jaka jest standardem w każdym innym przemyśle. A skoro nikt nie bierze odpowiedzialności, to klient nie ma wyboru.

W mojej ocenie przyczyną tego stanu rzeczy jest brak dostatecznie rozwiniętych modeli matematycznych, które pozwalałyby w procesie projektowania produktu programistycznego uzyskiwać gwarancje jego funkcjonalnej poprawności. Zresztą ten brak ma wpływ nie tylko na produkty z obszaru IT, ale też i na podręczniki języków programowania, co znów pośrednio wpływa na jakość produktów.

Zresztą w obszarze podręczników też nie widzę postępu, a nawet widzę regres. Wydany w roku 1960 definicyjny raport języka Algolu 60 [5] — języka, który w dużej mierze wytyczył drogę rozwoju informatyki dla kilku pokoleń programistów — wg. mnie znacznie przewyższał dzisiejsze podręczniki pod względem nie tylko precyzji i pełności wypowiedzi, ale też i zwięzłości².

Po pierwsze, składnia Algolu 60 była opisana nie jak to się czyni dziś, przez najczęściej niejasne przykłady, ale za pomocą *gramatyk generacyjnych*.

² Podobnymi cechami charakteryzował się wydany w pięć lat później podręcznik Algolu 60 w języku polskim [61].

Po drugie, semantyka języka, choć nie odwołująca się do modeli matematycznych (nie były wówczas znane), była opisana przy użyciu dobrze zdefiniowanego aparatu pojęciowego: *zmienna*, *blok*, *widoczność zmiennej*, *procedura*, *parametr procedury*, *rekurencja*.

Podobnymi cechami charakteryzował się powstały w około dziesięć lat później podręcznik języka Pascal [48]³. Niestety nie można tego powiedzieć o dzisiejszych podręcznikach, których autorzy nie odróżniają wyrażenia od instrukcji, a instrukcji od deklaracji.

Że zjawiska te są powszechne i dotyczą nie tylko języków programowania, ale także narzędzi budowania systemów, takich jak np. Joomla! czy Drupal, świadczą cieszące się niezmienną popularnością internetowe fora pomocowe, na których zdesperowani użytkownicy wymieniają się własnymi doświadczeniami. Do podręczników systemów mało kto zagląda, bo nie dość, że są nieprecyzyjne i niekompletne, to jeszcze dalece nieczytelne i to zarówno ze względu na swój język pozbawiony najczęściej aparatu pojęciowego, jak i na obszerność. Podczas gdy podręcznik Algolu 60 obejmował 237 stron, a podręcznika Pascala 166 stron, podręcznik Pytona [59] zawiera 696 stron, systemu Access [68] — 952 strony, a prawie zapomnianego już dziś języka Delphi, który miał się stać uniwersalnym językiem programowania wszechczasów, przekracza 2000 stron. Fora samopomocy wypełniają się więc pytaniami typu „Hej, czy ktoś wie jak...?”, na które zresztą dość często nikt nie udziela odpowiedzi. Średnio na trzy zadane przeze mnie pytania dwa pozostają bez odpowiedzi. Znajduję jedynie na forum podobne pytania innych użytkowników, które uświadamiają mnie, że z moim problemem nie jestem sam. Więcej na ten temat piszę w Aneksie 2.

³ A także opis Pascala w języku polskim [53].

1 Wprowadzenie

1.1 Odwróćmy tradycyjną kolej rzeczy

Problem pozyskiwania matematycznych gwarancji poprawności programów pojawił się po raz pierwszy w pracy Alana Turinga [66] opublikowanej w raporcie z konferencji On High Speed Calculating Machines, która miała miejsce na Uniwersytecie Cambridge w roku 1949. Później przez kilka dekad podejmowano ten temat pod hasłem *dowodzenia poprawności programów*, jednakże nigdy nie opracowano metod, które weszłyby na stałe do repertuaru narzędzi programistycznych. Wreszcie zaniechano tych prób, co w dobitny sposób podsumowali autorzy opublikowanej w roku 2016 monografii *Deductive Software Verification* [2] stwierdzając na str.2 co następuje (tłumaczenie moje):

Przez wiele lat pojęcie formalnej weryfikacji było niemalże synonimem weryfikacji funkcjonalnej⁴. W ostatnich latach staje się coraz bardziej jasne, że pełna funkcjonalna weryfikacja programu jest dla prawie wszystkich zastosowań celem iluzorycznym. Jak na ironię, dochodzimy do tego wniosku właśnie dzięki rozwojowi technik weryfikacyjnych: wraz z narodzeniem się systemów takich jak KeY, które w większości obejmują i precyzyjnie modelują rzeczywiste języki programowania, i które radzą sobie z większością systemów rzeczywistych, stało się ostatecznie oczywiste, jak bardzo trudne i czasochłonne jest tworzenie specyfikacji rzeczywistych systemów oprogramowania. Prawdziwym wąskim gardłem weryfikacji funkcjonalnej jest nie weryfikacja, ale specyfikacja⁵.

W mojej ocenie niepowodzenie prób zbudowania formalnego i praktycznego zarazem systemu uzyskiwania dowodów poprawności programów, a więc tzw. *logiki programów*, miało źródła w dwóch grupach problemów.

Pierwsza grupa wiąże się z faktem, że zbudowanie logiki programów dla konkretnego języka wymaga opisu semantyki tego języka na gruncie matematyki. Jednocześnie od lat 1970-tych dla matematyków było raczej oczywiste, że aby semantyka języka programowania miała praktyczny sens, musi być *kompozycyjna*, przez co rozumiemy, że znaczenie całości jest funkcją znaczeń jej części. Semantyki kompozycyjne nazwano *denotacyjnymi* (znaczenie programu to jego *denotacja*) i przez wiele lat badano ich właściwości teoretyczne oraz podejmowano próby budowania takich semantyk dla istniejących języków programowania. Wśród najbardziej

⁴ Przez „weryfikację funkcjonalną” autorzy rozumieją dowodzenie częściowej lub całkowitej poprawności programu, gdyż dotyczą one reprezentowanej przez program funkcji transformującej stany komputera. W swojej ponad 600 stron liczącej monografii zajmują się natomiast niefunkcjonalnymi własnościami programów takimi jak własność stopu, wykorzystania zasobów, przekroczenia zakresu, czy też wygenerowania błędu (obsługi wyjątków).

⁵ For a long time the term formal verification was almost synonymous with functional verification. In the last years it became more and more clear that full functional verification is an elusive goal for almost all application scenarios. Ironically, this happened because of advances in verification technology: with the advent of verifiers, such as KeY, that mostly cover and precisely model industrial languages and that can handle realistic systems, it finally became obvious just how difficult and time consuming the specification of functionality of real systems is. Not verification but specification is the real bottleneck in functional verification.

znanych dokonań praktycznych w tym obszarze należy wymienić formalne semantyki dla języków ADA [12] oraz CHILL [31] opisane w metajęzyku VDM [10]. Inny eksperyment z tej grupy, choć na znacznie mniejszą skalę, to definicja podzbioru Pascala [21] napisana w metajęzyku MetaSoft zbliżonym do VDM.

Niestety żadna z tych prób nie doprowadziła do wprowadzenia modeli denotacyjnych do praktyki inżynierii oprogramowania. W moim przekonaniu nie mogło być inaczej, gdyż dla istniejących języków sensownych semantyk denotacyjnych zbudować się raczej nie da (więcej na ten temat w rozdziale 4). Przyczyną tego stanu rzeczy jest fakt, że języki programowania budowano rozpoczynając od składni, a nie od denotacji. Innymi słowy najpierw zdecydowano, jak będziemy wyrażać treści, a dopiero później, czym te treści miałyby być. To oczywiście prosta konsekwencją historycznego faktu, że gdy zaczęto się zastanawiać nad tworzeniem semantyk, składnie języków już od dawna istniały.

Na te przyczyny nakładał się dodatkowo problem matematyczny polegający na tym, że w modelach denotacyjnych starano się opisać dwie konstrukcje programistyczne charakterystyczne dla języków algorytmicznych lat 1960-tych, które później zostały porzucone: instrukcję **goto** obecną w większości języków tamtego okresu oraz możliwość wysłania do procedury jej samej jako swojego parametru aktualnego (Algol 60, por. [61]). Pierwsze doprowadziły do pojęcia *kontynuacji* (por. [45]), a drugie do idei *dziedzin zwrotnych* (por. [63]). Jedne i drugie powodowały, że modele denotacyjne były technicznie złożone, a przez to mało czytelne, co zniechęcało do ich stosowania nie tylko praktyków, ale też wielu matematyków.

Druga grupa problemów wiąże się z milczącym założeniem, że budowanie programów matematycznie poprawnych powinno przebiegać w kolejności — najpierw program, a później dowód jego poprawności. Ta kolejność jest oczywista w matematyce — najpierw hipoteza, a później jej dowód — jednak dla każdego inżyniera absurdalny musi być pomysł, aby najpierw zbudować most, a dopiero później wykonać jego projekt wraz z obliczeniami.

Zasada „najpierw program, a później dowód poprawności” jest jednak nie tylko irracjonalna, ale też i praktycznie raczej niewykonalna, a to z dwóch powodów.

Po pierwsze, dowód jest z reguły dłuższy od twierdzenia, a twierdzenie o poprawności programu jest oczywiście dłuższe od samego programu. Skoro więc „praktyczne” programy liczą od dziesiątków tysięcy do milionów linii kodu, to „ręczne” dowodzenie ich poprawności nie wchodzi w grę. Z kolei praktycznych systemów dowodzenia automatycznego nie udawało się zbudować, choćby z tego powodu, że dla istniejących języków programowania nie stworzono matematycznych semantyk.

Jednakże wydaje mi się, że praktycznie ważniejszy jest powód drugi. Otóż programy, których poprawność staralibyśmy się udowodnić z reguły poprawne nie są! Stąd metody dowodzenia poprawności miały w zasadzie funkcjonować jako narzędzia wykrywania błędów. Gdy dowód poprawności się „zacina”, poprawiamy program, a następnie „uruchamiamy” dowód od nowa. Innymi słowy, najpierw robimy coś złe, by później to poprawiać. To chyba niezbyt racjonalne podejście.

Wobec wskazanych wyżej argumentów podejmuję w niniejszej książce próbę przedstawienia metod matematycznych pozwalających na zbudowanie języka programowania opartego na modelu denotacyjnym. Przykładowy język tego typu, który nazwałem **Lingua** — i który służy mi do ilustrowania ogólnych zasad na konkretnym przykładzie — buduję od denotacji do składni, co po raz pierwszy zaproponowałem w mojej pracy [22]. Zatem najpierw decydujemy, jakie chcemy wyrażać treści (denotacje), a dopiero później, jak te treści mają być zapisywane przez programistę (składnia). Zarówno denotacje jak i składnia języka są algebrami wielorodzajowymi (rozdział 2.11), a jego semantyka jest homomorfizmem ze składni w denotacje. Jak

się okazuje, istnieje prosta metoda — a nawet w dużym stopniu algorytmizowalna — pozwalająca dla zadanej algebry denotacji zbudować odpowiadającą jej składnię i semantykę.

Lingua w aplikatywnej warstwie struktur danych obejmuje wartości boolowskie, liczby, teksty, listy, rekordy i tablice (oraz dowolne ich kombinacje), a także SQL-owe bazy danych⁶. Jest również wyposażony w stosunkowo bogaty mechanizm typów, np. pozwalający na definiowanie SQL-owych więzów integralności, oraz w narzędzia do budowania przez programistę złożonych typów strukturalnych. Natomiast w warstwie imperatywnej jest wyposażony w typowe konstruktory programów strukturalnych, a w tym procedury z uwzględnieniem mechanizmów rekurencji i rekurencji wzajemnej wielu procedur. Wstępnie naszkicowałem też mechanizmy służące do programowania obiektowego.

Zagadnienia współbieżności zostały w książce pominięte, gdyż zdefiniowanie w tym przypadku semantyk „w pełni” denotacyjnych — jeżeli w ogóle możliwe — wymagałoby odrębnych badań wykraczających poza zakres niniejszej książki⁷.

Oczywiście nie buduję też mojego języka ze wszystkimi praktycznymi szczegółami, gdyż to uczyniłoby książkę nieczytelną. Pokazuję jednak — jak sądzę — dość uniwersalny model denotacyjny, który pozwala na opisanie wielu praktycznych narzędzi programowania sekwencyjnego.

Mając język zbudowany na denotacyjnym fundamencie wyposażam go w narzędzia do budowania *metaprogramów poprawnych*, których ideę naszkicowałem w pracy [18]. W skrócie są to programy, których teksty zawierają w sobie ich matematyczną specyfikację i które są względem tej specyfikacji poprawne (rozdział 8). Język takich programów jest wyposażony w reguły konstruowania metaprogramów poprawnych z poprawnych składowych. Programista uzyskuje więc gwarancję poprawności programu wprost ze sposobu, w jaki swój program buduje.

Wśród najważniejszych narzędzi matematycznych, jakimi posługuję się w mojej książce, należy wymienić:

1. teorię punktu stałego w zbiorach częściowo uporządkowanych,
2. rachunek relacji binarnych
3. teorię języków formalnych, a w tym gramatyki generacyjne w wersji równaniowej,
4. równania dziedzinowe budowane na gruncie tzw. *naiwnej semantyki denotacyjnej* (por. [29]),
5. teoria algebr wielorodzajowych,
6. technika błędów abstrakcyjnych pozwalająca na wbudowanie do systemu mechanizmu reakcji na semantyczne błędy użytkownika,
7. trójwartościowe rachunki predykatów wg. McCarthy’ego i Klennee’ego,

⁶ W Aneksie dodaję jeszcze drzewa, których nie umieściłem w „głównym nurcie” książki, by nie przeładować jej nadmiernie rozważaniami technicznymi.

⁷ W literaturze znane są semantyki dla współbieżności, o których można powiedzieć, że są „częściowo denotacyjne”. Jak przykład można podać semantyki oparte na składowych (ang. component-based semantics), gdzie denotacji są przypisywane programom w sposób kompozycyjny (tzn. że denotacja całości jest złożeniem denotacji ich części), natomiast denotacje są tzw. fukonami, których semantyka ma charakter operacyjny.

8. teorię całkowitej poprawności programów uwzględniającą nie tylko problem stopu, ale też „czystą terminację” programu, tj. zakończenie jego biegu bez wygenerowania sygnału błędu.

Te wszystkie narzędzia omawiam w rozdziałach 2 i 3, od czytelnika nie oczekuję więc w zasadzie żadnej ich znajomości. Wystarczy „obyć się” z podstawami matematyki w zakresie teorii mnogości i logiki matematycznej oraz znajomość podstaw programowania.

Budując **Lingua** przyjąłem też trzy priorytety związane z wyborem mechanizmów języka programowania:

- priorytet prostoty modelu, a więc prostoty denotacji, składni i łączącej je semantyki; np. rezygnacja z instrukcji **goto** oraz z procedur, które mogłyby brać same siebie jako parametry,
- priorytet prostoty reguł budowania poprawnych metaprogramów z poprawnych składowych; np. założenie, że deklaracje zmiennych i procedur oraz definicje typów występują jedynie na początku programów,
- priorytet ochrony przed przeoczeniami programisty, np. rezygnacja ze zmiennych globalnych we wszystkich procedurach oraz z efektów ubocznych w procedurach funkcyjnych.

Te trzy zasady doprowadziły mnie do świadomej rezygnacji z niektórych konstrukcji, które co prawda dałyby się matematycznie i denotacyjnie opisać, jednak prowadziłyby do złożonych opisów i jeszcze bardziej skomplikowanych reguł budowania programów. Priorytet prostoty i bezpieczeństwa przedkładam więc nad „elastyczność”, co zresztą już nie raz miało miejsce w historii języków programowania, np. rezygnacja z **goto** oraz z samoaplikowalnych procedur. Warto też podkreślić, że niektóre podejmowane w tym trybie decyzje — np. dwie właśnie wymienione — realizują wszystkie trzy priorytety.

Na koniec uwaga językowa. Czytelnik już pewnie zauważył, że wbrew dość powszechnemu zwyczajowi nie piszę o sobie w trzeciej osobie liczby pojedynczej, np. „zdaniem autora”, ale w pierwszej — „moim zdaniem”. Nie nadużywam też liczby mnogiej w rodzaju „w dalszym ciągu będziemy postępowali w analogiczny sposób”, bo to ja będę tak postępował, a nie „my z bożej łaski panujący ...” (tzw. „my” królewskie). Jeżeli używam „my”, to mam na myśli „ja i czytelnik”, a więc na przykład „zauważmy, że ...”. Postanowiłem przyjąć taką konwencję, by podkreślić, że niniejsza książka jest dla mnie tekstem bardzo osobistym w tym sensie, że nie przedstawiam w niej prawd objawionych, ale własne poglądy, z którymi nie wszyscy muszą się zgadzać.

Przyjąłem też zasadę dotyczącą pisowni słowa „czytelnik”. Piszę to słowo z dużej litery jedynie wtedy, gdy zwracam się do czytelnika, a więc np. „Zwróć uwagę Czytelniku na...”. Jednakże napiszę „Zwracam uwagę czytelnikowi...”.

1.2 Co zawiera książka

Jestem głęboko przekonany, że o programowaniu można mówić w sposób precyzyjny i zrozumiały zarazem. Wierzę również, że branie odpowiedzialności za produkt powinno być możliwe w przemyśle IT w równym stopniu jak jest to możliwe w przemyśle budowlanym, samochodowym czy lotniczym. Mam też jednak świadomość, że dostępne dziś narzędzia inżynierii programowania nie pozwalają na realizację takiego zamierzenia.

Jak już częściowo zapowiadałem, książka zawiera wiele myśli i idei rozwijanych w latach 1960-1990, które później zostały porzucone. W tamtych pionierskich czasach istniało na świecie zaledwie kilka zespołów naukowych budujących metajęzyki opisu systemów oprogramowania. Jeden z nich działał w Instytucie Podstaw Informatyki Polskiej Akademii Nauk, a ja miałem przyjemność nim kierować. Tworzony przez nas język nazwaliśmy **MetaSoft**, gdyż był to przede wszystkim język do opisu języków programowania, a więc *metajęzyk* (patrz [21]). Ten właśnie język przyjmuję jako narzędzie opisu modeli denotacyjnych języków programowania.

Moją książkę rozpoczynam od przypomnienia aparatu matematycznego związanego z **MetaSoft** (Rozdz. 2), który obejmuje wszystkie narzędzia wymienione w Rozdz. 1.1 poza teorią częściowej i całkowitej poprawności programów. Tę ostatnią buduję na gruncie rachunku relacji binarnych, co pozwala na abstrahowanie od technicznych szczegółów języków programowania (Rozdz. 3).

Rozdział 4 obejmuje ogólne omówienie idei denotacyjno-algebraicznego modelu języków programowania, który znajdzie zastosowanie przy budowaniu kolejnych warstw języka **Lingua**.

Rozdział 5 jest poświęcony budowie dość uniwersalnego modelu struktur danych i ich typów. Ten model starałem się potraktować na tyle szeroko, by dał się zastosować do dostatecznie dużej grupy algorytmicznych języków programowania włączając w to języki odwołujące się do standardu SQL. Jego denotacje, składnia i semantyka tworzą fundament aplikatywny dla pozostałych warstw języka. Ten fundament nazwałem **Lingua-A**.

Rozdział 6 zawiera model języka **Lingua-1** obejmujący mechanizm struktur danych, a więc obejmujący **Lingua-A**, a dodatkowo podstawowe konstruktory instrukcji, deklaracji zmiennych i definicji typów. Te ostatnie pozwalają programiście na strukturalne budowanie własnych typów, a także na ich zapamiętywanie przez nadawanie im nazw. **Lingua-1** jest pomyślany w ten sposób, by dał się rozbudowywać do modeli zawierających bogatsze mechanizmy programistyczne.

W rozdziale 7 wzbogacam **Lingua-1** do **Lingua-2**, który obejmuje rekurencyjne procedury imperatywne — a w tym z rekurencją wzajemną — oraz procedury funkcyjne.

Rozdział 8 jest poświęcony idei i technikom *programowania walidującego*, czym zajmowałem się na przełomie lat 1970/80. Jak już pisałem, podstawą tej idei jest pomysł, by zamiast dowodzić poprawność (z reguły niepoprawnych) programów, budować programy w sposób, który gwarantowałby ich poprawność. Tak tworzone programy, które nazwałem metaprogramami, zawierają w sobie dokumentację poprawnościową w postaci prewarunków, postwarunków i asercji. Reguły konstruowania metaprogramów gwarantują, że z poprawnych składowych powstają poprawne metaprogramy. Język programowania walidującego zbudowany nad językiem **Lingua-2** nazywam **Lingua-2V** (V od *validation*).

W rozdziale 9 szkicuję model dla języka z programowaniem obiektowym, w które wyposażam **Lingua-3**, a w rozdziale 10 bardzo krótko omawiam ideę obiektów zewnętrznych, a więc tworzonych poza **Lingua-3**.

Rozdziały 11 i 12 są poświęcone wyposażeniu języka **Lingua** w mechanizmy pozwalające na przetwarzanie SQL-owych baz danych. Tak rozbudowany język nazywam **Lingua-SQL**.

Zdaję sobie oczywiście sprawę, że to, co zawiera niniejsza książka, to dość ograniczony obraz bardzo bogatego dziś świata języków programowania. Od czegoś jednak trzeba zacząć. W **Lingua** umieściłem te konstrukcje, które są znane w informatyce od bardzo dawna i które nadal są dość powszechnie stosowane. W przyszłości postaram się uzupełnić moje modele o

obszary, które czytelnicy uznają za ważne. Liczę też, że niektórzy z Was sami podejmą to wyzwanie. Wszystkich zainteresowanych gorąco zapraszam do współpracy.

1.3 Czego książka nie oferuje

W Przedmowie i w Rozdz. 1.1 odniosłem się do występującego w przemyśle IT zjawiska, które skłoniło mnie do napisania niniejszej książki. Najogólniej rzecz ujmując, jest to brak narzędzi matematycznych, które pozwalałyby producentowi oprogramowania brać za swoje produkty odpowiedzialność w podobny sposób, jak czynią to producenci samochodów, telewizorów czy mostów. Nie oznacza to jednak, że moja książka oferuje gotowe narzędzia, które można by już dziś użyć w przemyśle oprogramowania. To co staram się zaproponować, to jedynie kierunek poszukiwania takich narzędzi oraz związanego z nimi aparatu matematycznego.

Aby wyjaśnić, co mam na myśli, odwołam się do pojęcia *jakości produktu* w takim sensie, w jakim jest dziś używane w tzw. *zarządzaniu kompleksową jakością*⁸. Otóż przez jakość produktu rozumiemy stopień spełnienia oczekiwań jego użytkownika. Tak zdefiniowaną jakość mierzy się liczbą wad w produkcji — im mniej wad tym wyższa jakość — przy czym wada to każda taka cecha produktu, która jest przez użytkownika niechciana, i której użytkownik ma się prawo nie spodziewać. Np. jeżeli zamawiamy w restauracji piwo, to mamy się prawo nie spodziewać, że będzie ciepłe, chyba że zamawiamy grzane.

Jakość produktu nie jest więc jego cechą immanentną, ale raczej relacją pomiędzy produktem, a tym, czego użytkownik oczekuje. Paradoksalnie można więc podnieść jakość produktu nie zmieniając tego ostatniego, ale opisując wszystkie jego niedostatki. Tego jednak wielu producentów oprogramowania nie robi, gdyż wtedy produkt mógłby się przestać sprzedawać.

W przypadku programów oczekiwanie użytkownika jest zapisywane w postaci specyfikacji, która ma być przez program spełniona. Na jakość programu składa się zatem:

1. zgodność specyfikacji programu z oczekiwaniami użytkownika,
2. zgodność programu ze specyfikacją.

W niniejszej książce zajmuję się jedynie drugim z tych aspektów. Ten wybór nie bierze się z przekonania, że problem pierwszy jest nieważny, ani tym bardziej, że został już rozwiązany, a jedynie stąd, że drugim zajmowałem się przed dwie dekady, na ten więc jedynie temat ośmielałem się dziś zabierać głos⁹.

Na koniec chciałbym podkreślić, że nie traktuję **Lingua** jako propozycji standardu, ani też praktycznego języka programowania, choć być może na jego kanwie taki język w przyszłości mógłby powstać. Na razie jest to jedynie przykład służący do ilustrowania opisywanych w książce konstrukcji i modeli. Starałem się w nim umieścić to, co w znanych mi językach programowania sekwencyjnego jest najczęściej spotykane.

1.4 Lingua z lotu ptaka

Dla ustrukturyzowania wykładu docelowy **Lingua** jest budowany w książce warstwami, o których była mowa w rozdziale 1.2. Poniżej przedstawiam mocno skondensowany i jedynie na

⁸ Zarządzanie kompleksową jakością, ang. Total Quality Management (skr. TQM) to dziś obszerna dziedzina wiedzy wykorzystywana w wielu przemysłach i branżach na całym świecie. Nauczam jej od roku 1997, a jej kompleksowe omówienie znajdzie czytelnik m.in. w mojej książce [27] dostępnej w wersji cyfrowej na witrynie <http://www.moznainaczej.com.pl/wydanie-drugie-dj/co-w-drugim-wydaniu>.

⁹ Myślę, że aspekt pierwszy jest równie fascynujący jak drugi, chętnie więc nawiązałbym współpracę z osobami, które byłyby gotowe popracować nad nim wraz ze mną.

wpół formalny opis tego języka bez wchodzenia w jego szczegóły techniczne, które są opisane w dalszych częściach książki. Nie opisuję też procesu budowania języka, a jedynie język docelowy. Ten rozdział jest adresowany przede wszystkim do osób, które chciałyby się zapoznać z podstawowymi elementami przedstawianego modelu bez czytania książki w całości.

W poniższym tekście posłużę się notacją, której pełny opis wraz z uzasadnieniem znajduje się w rozdziale 2.1.

- $a:A$ oznacza przynależność elementu a do zbioru A ,
- $f.a$ oznacza $f(a)$, natomiast $f.a.b.c$ oznacza $((f(a))(b))(c)$; intuicyjnie można powiedzieć, że funkcja f bierze a jako argument i zwraca wartość $f(a)$, która jest funkcją biorącą b jako argument i zwracającą funkcję $(f(a))(b)$, która bierze argument c i zwraca wartość $((f(a))(b))(c)$,
- $A \rightarrow B$ oznacza zbiór wszystkich *funkcji częściowych* z A do B , tj. takich które dla pewnych elementów zbioru A mogą (ale nie muszą) być nieokreślone,
- $A \rightarrowtail B$ oznacza zbiór wszystkich *funkcji całkowitych* z A do B , tj. takich które są określone dla wszystkich elementów zbioru A ; oczywiście każda funkcja całkowita jest szczególnym przypadkiem funkcji częściowej,
- $A \Rightarrow B$ oznacza zbiór wszystkich funkcji z A do B zdefiniowanych dla jedynie skończonych podzbiorów zbioru A ; takie funkcje nazywamy *mappingami* i one również stanowią przypadek szczególny funkcji częściowych,
- $A|B$ oznacza sumę teoriomnogościową A i B ,
- $A \times B$ oznacza iloczyn kartezjański A i B ,
- pp oraz ff oznaczają wartości logiczne „prawda” i „fałsz”,
- napisy typu dom , kor , kom oznaczają metazmienne przebiegające dziedziny, natomiast ujęty w apostrofy napis 'abcd' oznacza samego siebie, a więc metastałą.

1.4.1 Dane i ich typy

Dane w **Lingua** możemy podzielić umownie na trzy grupy:

- *dane proste* obejmujące dane boolowskie, liczby i słowa (skończone ciągi symboli),
- *dane strukturalne* obejmujące listy, tablice (wielowymiarowe), rekordy i drzewa oraz dowolne ich kombinacje,
- *dane SQL-owe* obejmujące wiersze i tabele przechowujące dane proste oraz bazy danych przechowujące tabele.

Dane strukturalne mogą „nieść” zarówno dane proste, jak i inne dane strukturalne. Możemy więc budować „głębokie” struktury danych, np. rekordy niosące listy drzew, w których węzłach stoją tablice. Listy i tablice składają się z elementów jednakowego typu, natomiast rekordy i drzewa nie są objęte tym ograniczeniem.

Listy i rekordy są zdefiniowane w dość tradycyjny sposób, z tym że elementami list, a także danymi przypisywanymi atrybutom rekordów mogą być dowolne dane proste lub strukturalne, choć nie SQL-owe.

Tablice są formalnie jednowymiarowe, ponieważ jednak elementami tablic mogą być inne tablice, możemy tworzyć tablice dowolnego wymiaru.

Drzewa są zdefiniowane jako pary składające się z rodzica i krotki dzieci, a więc są postaci (rodzic, (dziecko₁, ..., dziecko_n)). Zarówno rodzic, jak i dzieci, mogą być dowolnymi danymi prostymi lub strukturalnymi, a w tym i drzewami.

Bazy danych są, w pewnym uproszczeniu, rekordami tabel (funkcjami skończonymi z identyfikatorów w tabeli), tabele natomiast, znów w uproszczeniu — listami wierszy, a wiersze — rekordami niosącymi jedynie dane proste.

Dla tak określonych danych zostały zdefiniowane typowe konstruktory, mamy więc do czynienia z algebrą danych. W tym miejscu należy wyjaśnić, że cały przedstawiony w książce model denotacyjny opiera się na pojęciu algebry wielorodzajowej. Wprowadzenie do teorii takich algebr znajduje się w rozdziałach od 2.10 do 2.14.

Lingua jest wyposażony w dość bogaty mechanizm typów, który stanowi pewne rozszerzenie podobnych mechanizmów spotykanych w językach programowania. Przez mechanizm typów rozumiem narzędzia programistyczne pozwalające na definiowanie typów przez użytkownika oraz nadawanie im nazw dla późniejszego wykorzystania, czy to przy definiowaniu nowych typów, czy też przy deklarowaniu zmiennych (mechanizm typów jest opisany w rozdziale 5.2)

Typy są parami składającymi się *korpusu* i *jarzma*, a z każdym typem jest związany zbiór danych tego typu, który nazywamy *klanem typu*.

Intuicyjnie korpus opisuje „wewnętrzną strukturę danej” — np. czy jest to liczba, tablica, czy rekord — a formalnie jest kombinacją krotek i mappingów (funkcji o skończonych dziedzinach). Dane proste mają korpusy będące jednoelementowymi krotkami słów (patrz rozdział 2.1.4): ('boolowska'), ('liczba') i ('słowo'). Korpusy list i tablic mają postać odpowiednio ('L', korpus) lub ('T', korpus), gdzie korpus jest wspólnym korpusem wszystkich elementów listy/tablicy, a *inicjał* 'L' lub 'T' wskazuje, czy mamy do czynienia z listą, czy z tablicą. Korpus rekordowy ma postać ('R', rekord-korpusów), gdzie nieformalnie zapisanym przykładem rekordu korpusów może być:

imię ; ('słowo'),
nazwisko ; ('słowo'),
rok-urodzenia ; ('liczba'),
lata-nagród ; ('T', ('liczba')), (*)
pensja ; ('liczba'),
premia ; ('liczba')

Napisy stojące po lewych stronach średników są identyfikatorami, które w tym przypadku nazywamy *atrybutami*. W naszym przykładzie pierwsze trzy atrybuty mają korpusy proste, a czwarty — korpus tablicowy. Przedstawiony tu korpus rekordowy nazwijmy roboczo *pracownik*.

Z każdym korpusem *kor* jest kojarzony zbiór wszystkich danych o tym korpusie, który oznaczamy przez *KLAN-Kr.kor*. Funkcja *KLAN-Kr* jest definiowana indukcyjnie względem struktury korpusów. Np. dane o korpusie *pracownik* to rekordy o wymienionych w korpusie atrybutach, którym są przypisywane dane o typach wskazanych przez korpus. Np. atrybutowi *lata-nagród* są przypisywane tablice liczb.

Kolejnym ważnym pojęciem ze świata danych i typów jest *kompozyt* będący parą (*dan*, *kor*) składającą się z danej i jej korpusu tj. taką, że:

dan : *KLAN-Kr.kor*

Kompozyty powstają w wyniku wykonywania wyrażeń danologicznych (o których dalej). Wszystkie operacje na danych są w **Lingua** formalnie zdefiniowane jako operacje na kompozytach, co pozwala na wbudowanie w nie mechanizmów sprawdzania, czy „dostarczone” operacji argumenty są odpowiednich typów. Np. gdybyśmy na liście liczb chcieli umieścić słowo, to odpowiednia operacja wygeneruje komunikat błędu.

Dysponując pojęciem kompozytu możemy zdefiniować pojęcia *transferu* oraz *jarzma*. Transfery są jednoargumentowymi funkcjami przekształcającymi kompozyty w kompozyty, a jarzma są transferami o wartościach będących kompozytami boolowskimi, tj. kompozytami postaci (pp, ('boolowska')) oraz (ff, ('boolowska')). Transfery, a więc w tym i jarzma, mogą przyjmować jako wartości błędy abstrakcyjne (o których nieco niżej).

Tak więc jarzma są bliskie jednoargumentowym predykatom na kompozytach¹⁰. Przykładem jarzma opisującego własności kompozytów o korpusie pracownik może być nierówność:

pensja + premia < 10000,

która będzie spełniona, gdy jej argument jest kompozytem rekordowym zawierającym atrybuty pensja i premia, a związane z tymi atrybutami dane są liczbami spełniającymi powyższą nierówność. Tak rozumiane jarzma występują w SQL, choć nie są tam tak nazywane.

Transfery mają jedynie charakter pomocniczy i służą do zdefiniowania algebry, wśród elementów której znajdują się jarzma. Z każdym transferem tra, a więc i jarzmem, kojarzymy jego klan

KLAN-Tr.tra = (kom | tra.kom = (pp, ('boolowska')))

na który składają się kompozyty spełniające ten transfer. Oczywiście klany tranzytów nieboolowskich są puste.

Parę składającą się z korpusu i jarzma nazywamy *typem*. Ze względów matematycznych typy definiujemy nieco szerzej jako pary składające się z korpusu i dowolnego transferu. Z każdym typem typ = (kor, tra) kojarzymy jego klan KLAN-Ty.typ na który składają się te kompozyty, których dane należą do klanu KLAN-Kr.kor, a one same (czyli kompozyty) spełniają transfer. Zatem:

KLAN-Ty.(kor, tra) = {(dan, kor) | dan : KLAN-Kr.kor **oraz** (dan, kor) : KLAN-Tr.tra}

Ostatnim pojęciem związanym z danymi i typami jest *wartość* zwana również *daną utypowaną*, która jest parą (dan, typ), czyli (dan, (kor, tra)), co można też zapisać jako ((dan, kor), tra). Na wartości można więc patrzeć jako na pary *dana-typ* lub na pary *kompozyt-transfer*.

Wartości są przypisywane w stanach identyfikatorom zmiennych, przy czym instrukcja przypisania — a więc nadawania zmiennej wartości — może zmienić jedynie daną oraz w ograniczonym zakresie jej korpus, ale nie jarzmo. Do zmiany jarzma służy odrębna instrukcja.

Podsumujmy więc listę obiektów występujących w obszarze danych i typów:

- *dane* to podstawowe obiekty przetwarzane przy wykonywaniu programów,
- *korpusy* to obiekty opisujące „struktury wewnętrzne” danych,
- *kompozyty* to pary (dan, kor), gdzie dan : KLAN-Kr.kor, które powstają w wyniku wykonywania wyrażeń *danologicznych*,

¹⁰ Są bliskie predykatom, a nie po prostu są predykatami, gdyż po pierwsze jako wartości przyjmują boolowskie kompozyty, a nie pp lub ff, a po drugie, przyjmują również błędy, a ich konstruktory logiczne **oraz**, **lub** i **nie** pochodzą z trójwartościowego rachunku zdań McCarthy'ego (rozdział 2.9).

- *transfery* to jednoargumentowe funkcje na kompozytach, a *jarzma* to transfery o wynikach będących kompozytami boolowskimi lub błędami,
- *typy* to pary składające się z korpusu i transferu (a w zasadzie z jarzma); jak zobaczymy dalej, typy powstają w wyniku wykonywania *wyrażeń typologicznych*, a w stanach są przypisywane *stałym typologicznym*,
- *wartości* to pary składające się z danej i (jej) typu; w stanie wartości są przypisywane identyfikatorom *zmiennych danologicznych*.

Podobnie jak w wielu językach programowania (choć nie we wszystkich), typy służą w **Lingua** czterem celom:

1. określaniu typu zmiennej w chwili jej deklaracji i zadbania, by w trakcie wykonywania programu ten typ pozostawał niezmienny (z pewnymi wyjątkami)¹¹,
2. zadbania, aby dana wiązana ze zmienną przez instrukcję przypisania była typu zgodnego z typem zmiennej,
3. zadbania, aby analogiczna zgodność miała miejsce przy przesyłaniu parametrów aktualnych do procedury i zwracaniu parametrów referencyjnych,
4. zadbania, aby przy wyliczaniu wartości wyrażeń był sygnalizowany błąd ilekroć wyrażeniu zostaną „dostarczone” dane nieodpowiednich typów; np. gdybyśmy próbowali dodać słowo do liczby.

1.4.2 Błędy abstrakcyjne

Charakterystyczną cechą języka **Lingua** jest uwzględnienie w jego modelu sytuacji związanych z pojawianiem się komunikatów błędów. W tym celu do dziedzin (zbiorów) korpusów, kompozytów i typów dodajemy elementy, które nazywamy *błędami*. Z czysto matematycznego punktu widzenia błędy mogą być czymkolwiek, natomiast w **Lingua** są to napisy, na przykład:

‘dzielenie-przez-zero’

‘oczekiwany-rekord’

które w intuicyjny sposób opisują przyczynę pojawienia się komunikatu błędu. Wszystkie operacje na korpusach, kompozytach i typach są zdefiniowane również na błędach, przy czym najczęściej mają one własność *transparentności względem błędów*, co oznacza, że gdy na „wejściu” otrzymają argument niosący błąd, to na „wyjściu” pojawi się ten sam błąd. Intuicyjnie oznacza to, że ilekroć w czasie wyliczania wyrażenia, lub wykonywania instrukcji pojawi się komunikat błędu, to ten komunikat jest wyświetlany na monitorze, a wykonanie programu zostaje przerwane. Może być jednak i tak, że pojawienie się określonego błędu powoduje uruchomienie procedury obsługi tego błędu (rozdziały 6.1.8 i 12.7.6.4).

W kontekście błędów szczególny przypadek stanowią operacje boolowskie (rozdział 2.9), które na błędy reagują zgodnie z rachunkiem zdań McCarthy’ego, a więc na przykład:

ff oraz bb = ff

bb oraz ff = bb

gdzie bb reprezentuje błąd lub niekończące się obliczenie. Argumenty koniunkcji są wyliczane od lewej do prawej i jeżeli pierwszym argumentem jest ff to drugiego argumentu już nie

¹¹ Te wyjątki mają miejsce, gdy np. do rekordu lub tabeli SQL-owej dodajemy nowy atrybut lub istniejący atrybut usuwamy.

obliczamy być może unikając w ten sposób pojawienia się komunikatu błędu lub wejścia w nieskończone obliczenie. Na przykład wyrażenie boolowskie

$$x \neq 0 \text{ oraz } 1/x > 10$$

dla $x=0$ przyjmie wartość ff, mimo że $1/x > 10$ wygenerowałoby błąd lub weszło w nieskończone obliczenie. Taki efekt się jednak nie pojawi, bo przy $x = 0$ nie dojdzie do wyliczania wartości $1/x > 10$.

Jednymi z sytuacji, w których pojawiają się błędy, są przekroczenia zakresu. Zakłada się więc, że wszystkie rodzaje danych (poza boolowskimi) mają przypisane sobie zakresy reprezentowalności w pamięci komputera wyrażone przez predykaty „reagujące” na przekroczenie zakresu.

1.4.3 Wyrażenia

Wyrażenia są obiektami składniowymi, a ich *denotacje*, czyli semantyczne znaczenia, są funkcjami, które stany pamięci odwzorowują w kompozyty (wyrażenia danologiczne) lub typy (wyrażenia typologiczne). Dla zdefiniowania tych pojęć należy więc zacząć od zdefiniowania dziedziny stanów za pomocą tzw. *równań dziedziny*:

Stan = Środ x Skład (stan)

Środ = ŚroTyp x ŚroPro (środowisko)

Skład = Wart x (Błąd | {'OK'}) (skład)

Wart = Identyfikator $\Rightarrow$ Wartość (wartościowanie)

ŚroTyp = Identyfikator $\Rightarrow$ Typ (środowisko typów)

ŚroPro = Identyfikator $\Rightarrow$ Procedura | Funkcja (środowisko procedur)

Jak widzimy, stany przechowują wartości, typy, procedury i funkcje (procedury funkcyjne) przypisywane identyfikatorom, a także błędy w „odrębnym rejestrze”. Jeżeli stan nie niesie błędu, to w tym rejestrze znajduje się napis 'OK'.

Denotacje wyrażeń danologicznych i denotacje wyrażeń typologicznych są elementami algebry denotacji wyrażeń, z której „powstaje” składniowa algebra wyrażeń. Homomorfizm z algebry wyrażeń w algebrę denotacji wyrażeń stanowi semantykę wyrażeń.

Denotacje wyrażeń danologicznych są funkcjami częściowymi ze stanów w kompozyty lub komunikaty błędu:

$$\text{DenWyrDan} = \text{Stan} \rightarrow \text{Kompozyt} \mid \text{Błąd}$$

gdzie Błąd jest zbiorem błędów. Każdej operacji na danych odpowiada operacja na kompozytach, której z kolei odpowiada konstruktor denotacji wyrażeń danologicznych. Na przykład denotacją wyrażenia

$$x + y$$

jest funkcja, która dla zadanego stanu sprawdza kolejno:

- czy stan nie niesie błędu?
- czy identyfikatorom x oraz y zostały przypisane wartości?
- czy są to wartości liczbowe?
- czy ich suma nie przekracza dopuszczalnej wielkości?

i jeżeli te wszystkie sprawdzenia kończą się pozytywnie, to tworzy kompozyt (`dan`, ('liczba')), gdzie `dan` jest sumą liczb przypisanych do x i y . Jeżeli którekolwiek sprawdzenie nie zakończy się pozytywnie, to dodawanie generuje odpowiedni błąd, np.

`'oczekiwana-liczba'`

i obliczenie się na tym kończy. Jeżeli natomiast stan wejściowy obliczenia niesie komunikat błędu, to ten komunikat staje się wynikiem obliczenia.

Podkreślam, że wyrażenia danologiczne reprezentują funkcje częściowe, gdyż dopuszcza się wśród nich wywołania procedur funkcyjnych, które mogą prowadzić do efektu zapętlenia, a więc niekończącego się obliczenia.

Denotacje wyrażeń typologicznych są funkcjami całkowitymi ze stanów w typy lub komunikatów błędów:

$\text{DenWyrTyp} = \text{Stan} \mapsto \text{Typ} \mid \text{Błąd}$

Konstruktory takich denotacji są zdefiniowane analogicznie jak uprzednio, z tym oczywiście, że odwołują się do operacji na typach, a nie na danych. Na przykład denotacją wyrażenia typologicznego:

record-type

```
imię          as word,
nazwisko      as word,
rok-urodzenia as number,
lata-nagród   as tablica-liczb,
pensja        as number,
 premia       as number
```

ee

jest funkcja na stanach, która tworzy typ rekordowy lub generuje błąd. To wyrażenie odwołuje się do dwóch typów systemowych `word` oraz `number` oraz jednego zdefiniowanego przez użytkownika `tablica-liczb`.

Denotacje wyrażeń danologicznych i typologicznych wraz z ich konstruktorami tworzą wspólną dla nich *algebrę denotacji wyrażeń*, która jest opisana w rozdziale 5.3. Odpowiadająca jej składniowa *algebra wyrażeń* jest opisana w rozdziale 5.4

1.4.4 Instrukcje

Wyrażenia należą do *aplikatywnej części języka*. Ich denotacje przyjmują stany jako argumenty, jednak ich ani nie tworzą, ani nie zmieniają. Te zadania realizują *instrukcje*, *deklaracje zmiennych*, *deklaracje procedur i funkcji* oraz *definicje typów*. Wszystkie one należą do *imperatywnej części języka*.

Denotacjami instrukcji są funkcje częściowe ze stanów w stany:

$\text{DenIns} = \text{Stan} \rightarrow \text{Stan}$

W odróżnieniu od denotacji wyrażeń, które mogą generować błąd, denotacje instrukcji wpisują błąd do rejestru błędu będącego składową stanu. Większość instrukcji ma denotacje transparentne względem stanów niosących błąd, tj. niezmienniające takich stanów, a jedynie

przekazujące je do następnych części programu. Niekiedy jednak błąd powoduje podjęcie specjalnej akcji, o czym była już mowa wyżej.

Podstawową instrukcją jest oczywiście *przypisanie* wartości do identyfikatora zmiennej. Składniowo przypisanie ma postać:

```
identyfikator := wyrażenie_danologiczne
```

Denotacja przypisania przekształca stan wejściowy w wyjściowy następujących krokach:

1. sprawdzenie, czy stan wejściowy nie niesie błędu, a jeżeli niesie, to staje się on stanem wyjściowym i na tym wykonanie się kończy; w przeciwnym przypadku,
2. sprawdzenie, czy identyfikator został zadeklarowany, tj. czy w stanie wejściowym jest mu przypisana wartość lub pseudowartość (o czym niżej); jeżeli tak nie jest to do rejestru błędu jest ładowany komunikat 'identyfikator-niezadeklarowany'; w przeciwnym przypadku,
3. próba wyliczenia wyrażenia; jeżeli w wyniku otrzymujemy błąd, to jest on ładowany do rejestru błędu, w przeciwnym przypadku,
4. sprawdzenie, czy kompozyt wyliczony z wyrażenia ma korpus zgodny z korpusem typu identyfikatora; jeżeli są niezgodne, to następuje załadowanie komunikatu błędu do rejestru, a w przeciwnym przypadku,
5. sprawdzenie, czy kompozyt wyliczony z wyrażenia spełnia jarzmo będące składową typu identyfikatora; jeżeli nie spełnia, to następuje załadowanie komunikatu błędu do rejestru, a w przeciwnym przypadku,
6. przypisanie wyliczonego kompozytu identyfikatorowi z pozostawieniem jarzma nienaruszonym.

Pozostałe instrukcje należą do jednego z sześciu niżej wymienionych rodzajów, z których pierwsze trzy to *instrukcje atomowe*, a pozostałe to *instrukcje strukturalne*, a więc złożone z instrukcji atomowych.

1. zmiana jarzma przypisanego identyfikatorowi w jego typie,
2. uruchomienie obsługi błędu,
3. wywołanie procedury imperatywnej,
4. sekwencyjne złożenie instrukcji,
5. warunkowe złożenie instrukcji typu **if-then-else-fi**,
6. pętla **while-do-od**.

Spośród tych rodzajów instrukcji omówienia wymaga w zasadzie jedynie wywołanie procedury imperatywnej.

Wywoływana procedura otrzymuje dwie listy *parametrów aktualnych* — parametry *wartościowe* i *referencyjne* — których wartości są przypisywane odpowiadającym im *parametrom formalnym*, również wartościowym i referencyjnym. Po wykonaniu treści wywołanej procedury wartości końcowe formalnych parametrów referencyjnych są przekazywane do aktualnych parametrów referencyjnych w stanie wynikowym wywołania.

Mechanizm parametrów stanowi jedyny kanał przesyłania wartości pomiędzy zewnątrz a wnętrzem procedury, co oznacza, że w ciele procedury nie są widoczne zmienne zadeklarowane na zewnątrz. Należy podkreślić, że to ograniczenie nie jest w żaden sposób „wymuszone” przez fakt, że model **Lingua** ma charakter denotacyjny. Przyjąłem je — jak i wiele innych

podobnych założeń — ze względów, które uznałem za pragmatyczne, co staram się uzasadnić w dalszej części książki.

W odróżnieniu od zmiennych, typy i procedury zadeklarowane w preambule programu (patrz dalej) mają charakter globalny, są więc dostępne w ciele każdej procedury. W treści procedury mogą oczywiście występować deklaracje lokalnych zmiennych i procedur, a także definicje lokalnych typów, nie są one jednak widoczne na zewnątrz procedury. To też decyzja inżynierska, a nie denotacyjna konieczność.

Dla procedur imperatywnych jest dostępny mechanizm rekurencji zarówno bezpośredniej (procedura wywołuje samą siebie), jak i pośredniej (procedura A wywołuje procedurę B, która wywołuje C ... która wywołuje A).

Denotacje wyrażeń danologicznych i typologicznych, instrukcji, deklaracji zmiennych i deklaracji procedur, definicji typów, preambuł i programów (o dwóch ostatnich dalej) tworzą wspólna dla nich wszystkich wielorodzajową algebrę denotacji, która jest opisana w rozdziałach 6 (bez procedur) oraz 7 (z procedurami). Tamże są opisane odpowiednie algebry składniowe języka **Lingua**.

1.4.5 Deklaracje zmiennych i definicje typów

Denotacje deklaracji zmiennych są funkcjami całkowitymi odwzorowującymi stany w stany:

$$\text{DenDekZmi} = \text{Stan} \mapsto \text{Stan}$$

przypisując identyfikatorowi typ i pozostawiając daną nieokreśloną. Bardziej formalnie, przypisują identyfikatorowi *pseudowartość*, która jest parą postaci (Ω, typ) , gdzie Ω jest bytem abstrakcyjnym, który nazywamy *pseudodaną*. Składniowo pojedyncza deklaracja zmiennej jest postaci:

```
let identyfikator be wyrażenie_typologiczne tel
```

Wykonanie deklaracji jest podobne do przypisania z tym, że tym razem błąd jest sygnalizowany, gdyby w stanie wejściowym był identyfikatorowi przypisany jakikolwiek byt — pseudowartość, wartość, typ lub procedura. W takim przypadku jest generowany komunikat błędu 'identyfikator-zajęty'. Jak stąd wynika, każda zmienna może być zadeklarowana w programie tylko raz. Później można jedynie zmieniać przypisany jej kompozyt i ewentualnie jarzmo. Korpus można zmienić jedynie w przypadku rekordów i bazodanowych tabel i jedynie wtedy, gdy jest dodawany nowy atrybut lub jakiś atrybut jest usuwany (decyzja inżynierska, a nie matematyczna konieczność).

Definicje typów są podobne do deklaracji zmiennych, z tym że identyfikatorom przypisują typy, a nie pseudowartości.

$$\text{DenDefTyp} = \text{Stan} \mapsto \text{Stan}$$

Składnia takich definicji jest następująca.

```
set identyfikator be wyrażenie_typologiczne tes
```

Identyfikator, któremu w stanie został przypisany typ, nazywamy *stałą typologiczną*. Właśnie „stałą”, a nie „zmienną”, bo przypisany jej typ przez cały czas wykonywania programu pozostaje niezmienny (decyzja inżynierska).

Definicje typów i deklaracje zmiennych można łączyć ze sobą złożeniem sekwencyjnym analogicznym do składania instrukcji.

1.4.6 Procedury

Procedury dzielimy na imperatywne i funkcyjne. Te pierwsze są funkcjami, które po otrzymaniu dwóch list parametrów aktualnych — wartościowych i referencyjnych — zwracają funkcje częściowe modyfikujące składy¹². Procedury funkcyjne po otrzymaniu listy aktualnych parametrów wartościowych — parametrów referencyjnych procedury funkcyjne nie mają — zwracają funkcje częściowe przekształcające stany w kompozyty:

$\text{pri} : \text{Prolmp} = \text{ParAkt} \times \text{ParAkt} \mapsto \text{Skład} \rightarrow \text{Skład}$

$\text{prf} : \text{ProFun} = \text{ParAkt} \mapsto \text{Stan} \rightarrow (\text{Kompozyt} \mid \text{Błąd})$

W powyższych równaniach ParAkt jest dziedziną *list parametrów aktualnych*. Zauważmy, że mówimy tu o procedurach, a nie o denotacjach procedur, bo procedury są bytami „czysto denotacyjnymi”, co oznacza, że nie mają odpowiedników składniowych. Na poziomie składni występują jedynie *deklaracje procedur* i *wywołania procedur* i one oczywiście mają swoje denotacje. Deklaracja procedury imperatywnej ma składniową postać:

```
proc identyfikator (val par_for_war ref par_for_ref)
    program
end proc
```

gdzie `program` jest programem (o czym nieco dalej). Listy parametrów formalnych i aktualnych są listami identyfikatorów. Wyrażenia inne niż pojedyncze identyfikatory nie są dopuszczane na miejscu parametrów wartościowych, gdyż to komplikowałoby zarówno model, jak i reguły konstrukcji programów poprawnych (decyzja inżynierska).

Jeżeli chcemy zadeklarować grupę wzajemnie rekurencyjnych procedur, to stosujemy *deklarację multiprocedury* postaci:

```
begin multiproc
    DekPro-1;
    DekPro-2;
    ...
    DekPro-n
end multiproc
```

Procedury funkcyjne są funkcjami częściowymi, które przekształcają stany (a nie składy!), odpowiadają więc wyrażeniom. One też nie mają odpowiedników po stronie składni. Ich deklaracje są postaci:

```
fun identyfikator (par_formalne)
    program
return wyrażenie as wyr_typologiczne
```

Wywołanie procedury funkcyjnej powoduje wykonanie zawartego w niej programu, a następnie wyliczenie wartości wyrażenia (tj. kompozytu) w stanie końcowym tego programu. Wywołanie procedury funkcyjnej nie ma więc tzw. *efektów ubocznych* w postaci modyfikacji stanu

¹² To że procedury działają na składach, a nie na stanach, jest zabiegiem technicznym pozwalającym uniknąć efektu samoaplikacji funkcji, tj. sytuacji gdy funkcja bierze samą siebie jako swój argument. Więcej na ten temat w rozdziale 4.1. Oczywiście wywołania procedur są instrukcjami, a więc przekształcają stany w stany.

(decyzja inżynierska). W szczególnym przypadku program może być trywialny, tj. nie zmieniać stanu, a wyrażenie może być pojedynczym identyfikatorem.

W rozdziale 7 opisano procedury, które jak parametry aktualne przyjmują jedynie identyfikatory zmiennych, a więc identyfikatory wiążące wartości. Zdefiniowane w programie typy i procedury są w ciele deklaracji dostępne jako byty globalne, nie muszą więc być przekazywane jako parametry (decyzja inżynierska). Jednakże opisany tu mechanizm nie przewiduje procedur, które mogłyby inne procedury nie tylko wywoływać, ale też przyjmować jako parametry. W modelu denotacyjnym jest to możliwe, jednak trzeba zbudować hierarchę typów proceduralnych, aby uniknąć sytuacji, w której procedura mogłaby przyjąć samą siebie jako parametr. W tym przypadku mamy więc do czynienia z sytuacją, w której decyzja dotycząca mechanizmów języka jest wymuszona przez denotacyjny charakter naszego modelu¹³. Ten temat jest omówiony skrótowo w rozdziale 7.6.

1.4.7 Programowanie obiektowe

Narzędzia projektowania obiektowego są w książce jedynie naszkicowane, aby wskazać ogólne zasady według jakich można budować denotacyjny model dla tych narzędzi. Najkrócej rzecz ujmując, *obiekt* — który podobnie jak procedura jest bytem jedynie po stronie denotacji — jest transformacją stanów zagnieżdżającą w nich pewną liczbę procedur, funkcji i typów. Składniowo więc *definicja obiektu* jest postaci:

```
set-object identyfikator as wyr_obiektowe tes-object
```

gdzie `wyr_obiektowe` jest sekwencja definicji typów i deklaracji procedur imperatywnych i funkcyjnych.

Obiekty są przechowywane w wydzielonej części pamięci, którą nazwałem *biblioteką obiektów*. Udostępnianie obiektu w programie dokonuje się za sprawą instrukcji *przywołania obiektu*. Przywołania obiektów mogą występować nie tylko w programach, ale też w definicjach obiektów, co w pewien sposób realizuje mechanizm dziedziczenia obiektów. Po nieco więcej szczegółów odsyłam czytelnika do rozdziałów 8.7 i 10.

1.4.8 Programowanie SQL-owe

W typowych językach programowania udostępniających bazodanowe narzędzia standardu SQL — znanych w literaturze anglojęzycznej jako *Application Programming Interface* lub też *Call Level Interface* — interpreter języka programowania odwołuje się do jakiegoś istniejącego już silnika SQL uruchamiając jego procedury. W naszym przypadku taka filozofia nie byłaby jednak do przyjęcia. Skoro bowiem język **Lingua** chcemy wyposażać w reguły budowania programów poprawnych, o których musimy udowodnić, że tę poprawność gwarantują, to dla SQL musimy zbudować własną implementację opartą na modelu denotacyjnym. Tak więc ewentualna implementacja **Lingua-SQL** nie może umożliwiać sięgania z programów pisanych w **Lingua** do jakiegoś silnika bazodanowego, ale musi być wyposażona we własny silnik.

Oczywiście należy zadbać o to, aby konstruktory bazodanowe zdefiniowane w naszym modelu były możliwie blisko temu, co jest uznane za standard SQL. O stuprocentowej zgodności nie może być jednak mowy, bo po pierwsze nie ma jednego standardu SQL, a po drugie, żaden z tych, które są, nie jest na tyle dobrze zdefiniowany, aby jego semantykę można było opisać

¹³ W rzeczywistości jest wymuszona względami związanymi bardziej z teorią mnogości niż z modelami denotacyjnymi. Więcej na ten temat w rozdziale 4.1.

jakimkolwiek modelem matematycznym. Należy więc zadbać jedynie o to, aby programy w **Lingua-SQL** mogły przetwarzać bazy danych stworzonych przez inne aplikacje SQL-owe.

Rozdział 12 zawiera model obejmujący denotacyjne definicje dla wybranych SQL-owych mechanizmów służących do tworzenia i przetwarzania baz danych. W szczególności obejmuje też zapytania, kursory i perspektywy.

1.4.9 Programy

Programy w **Lingua** składają się z dwóch następujących po sobie części:

1. *preambuły*, która zawiera dowolną liczbę sekwencyjnie złożonych deklaracji zmiennych, deklaracji procedur i funkcji, definicji typów i przywołań obiektów.
2. instrukcji, która oczywiście może być dowolnie złożona.

Ta struktura programu — najpierw wszystkie deklaracje i definicje, a później wszystkie instrukcje — nie jest w żaden sposób „wymuszona” przez denotacyjny model języka, a wynika jedynie z zamiaru stworzenia możliwie prostych reguł budowania programów poprawnych.

1.4.10 Programowanie walidujące

Programowanie walidujące polega na budowaniu *metaprogramów* składających się z dwóch przeplatających się ze sobą warstw:

1. warstwy *programistycznej*, czyli programu właściwego,
2. warstwy *deskryptywnej*, na którą składają się prewarunek i postwarunek oraz stojące wewnątrz programu asercje odpowiadające warunkom.

O tak rozumianym metaprogramie mówimy, że jest *poprawny*, jeżeli dla każdego stanu, który spełnia prewarunek,

1. program się wykona bez zapętlania i wygenerowania błędu,
2. wszystkie asercje będą spełnione w trakcie wykonywania programu,
3. stan końcowy programu spełni postwarunek.

Mamy więc do czynienia z całkowitą poprawnością w sensie C.A.R. Hoare’a, przy czym jest to poprawność rozumiana nieco szerzej, bo w logice Hoare’a nie są uwzględnione błędy abstrakcyjne, a więc problem „zawieszania się” programu.

Zwróćmy uwagę, że o ile w klasycznej teorii poprawności programów mówimy o poprawności programu względem zadanych prewarunku i postwarunku, o tyle w naszym przypadku mówimy o poprawności jako takiej, bo prewarunek i postwarunek stanowią składowe metaprogramu. Włączenie warstwy deskryptywnej do metaprogramów pozwala na zdefiniowanie reguł budowania złożonych metaprogramów poprawnych z ich poprawnych składowych.

Poniżej przykład prostego metaprogramu, gdzie $isr(n)$ oznacza część całkowitą pierwiastka kwadratowego z n :

```
Q4: let z, x be number
    pre true                                     (prewarunek)
    z := 1;
    x := 0;
    begin-asr z > 0 and x ≥ 0                    (zadeklarowanie warunku)
```

```

    while  $z^2 \leq n$  do  $z := 2 * z$  od;
    x := 0;
    while z > 1
    do
        z := z/2;
        if  $(x+z)^2 < n$  then  $x := x+z$  else  $x := x$  fi
    od
end-asr
post x = isr(n) and z = 1

```

(odwołanie deklaracji warunku)
(postwarunek)

Obszar programu pomiędzy **begin-asr** war oraz **end-asr** nazywam *obszarem obowiązywania warunku* war. Jeżeli metaprogram jest poprawny, to dekretowany warunek musi być spełniony przez wszystkie stany pośrednie w obszarze jego obowiązywania.

Konstruowanie programu poprawnego rozpoczyna się od prostych programów, których poprawność jest dowiedziona w sposób tradycyjny. Kolejne programy są tworzone z wcześniej utworzonych za pomocą reguł gwarantujących zachowanie poprawności. A oto przykład takiej reguły pozwalającej na zbudowanie poprawnej instrukcji będącej rozgałęzieniem warunkowym:

```

(1) prw  $\Rightarrow$  wyd or (not wyd)
(2) def pab pre (prw and wyd) ins-1 post pow
(3) def pab pre (prw and not wyd) ins-2 post pow
↓
def pab pre prw if wyd then ins-1 else ins-2 fi post pow

```

Tę regułę czytamy w sposób następujący:

Jeżeli

- (1) każdy stan, który spełnia prewarunek prw, spełnia wyrażenie danologiczne wyd lub jego negację; to założenie oznacza że przy spełnionym prewarunku, wyrażenie danologiczne wyd stojące w rozgałęzieniu wylicza się do wartości boolowskiej, a więc w szczególności nie generuje błędu i nie zapętiła się,
- (2) występujący tu metaprogram jest poprawny,
- (3) występujący tu metaprogram jest poprawny,

to

poprawny jest metaprogram występujący pod kreską.

Ta reguła pozwala na zbudowanie metaprogramu poprawnego z dwóch metaprogramów poprawnych. Zauważmy, że w klasycznym dwuwartościowym rachunku predykatów, (1) byłby tautologią, jednak w przypadku predykatów mogących zwracać błędy, już tak nie jest.

2 MetaSoft i jego matematyka

Gdy w latach od 1970 do 1990 prowadziłem wykłady z matematycznych podstaw informatyki dla praktyków, nieraz słyszałem obiekcję, że tej nowej matematyki, której trzeba się nauczyć, jest stanowczo zbyt wiele. Szefowie zespołów IT oczekiwali, aby w tym zakresie ich pracowników „przeszkolić” w weekend, no może w dwa weekendy. Tyle możemy na to poświęcić — mówili — więcej nie. Wtedy uświadamiałem im, ile czasu poświęca na naukę matematyki przyszły inżynier kończący studia z dyplomem politechniki. W zależności od wydziału jest to od dwóch do pięciu semestrów. Ta jednak matematyka, stworzona najczęściej w wieku XIX i na początku wieku XX, powstała na użytek astronomii i fizyki, a później też innych nauk przyrodniczych, a także społecznych (ekonomia i socjologia), ale nie na użytek informatyki.

Gdy na początku drugiej połowy XX wieku zaczęto myśleć o zastosowaniu matematyki do opisu zjawisk i obiektów informatycznych, zastosowania znalazły teorie matematyczne uznawane wcześniej za dalece „niepraktyczne” takie jak teoria mnogości, logika matematyczna, czy teoria algebr abstrakcyjnych. Z czasem też zaczęły powstawać teorie budowane dla specyficznych obszarów informatyki: teoria automatów abstrakcyjnych, języków formalnych, systemów współbieżnych, różnego rodzaju logiki algorytmiczne służące do wyrażania i dowodzenia własności programów. Dziś tej matematyki starczy nie na pięć, ale na dziesięć semestrów (lub na więcej), a jest ona cały czas rozwijana pod wspólną nazwą *matematycznych podstaw informatyki*.

W niniejszym rozdziale nie podejmuję oczywiście zadania przedstawienia całej tej matematyki, choćby w zarysie. Ograniczam się jedynie do tych narzędzi, które będą mi potrzebne przy budowaniu **Lingua**. Mam też świadomość, że dla niektórych czytelników przebrnięcie przez rozdział 2 będzie wymagało pewnego trudu, jednakże opanowany w ten sposób metajęzyk¹⁴ — który w latach 1980. otrzymał nazwę **MetaSoft** — pozwoli złożone obiekty informatyczne opisywać w sposób stosunkowo prosty, oraz — co szczególnie ważne — pełny i jednoznaczny.

Zakładam, że czytelnikowi są znane podstawowe pojęcia związane z rachunkiem zbiorów (teorią mnogości) oraz elementy logiki matematycznej w zakresie rachunku zdań i predykatów.

2.1 Podstawowe konwencje notacyjne MetaSoft

MetaSoft, w wersji w jakiej go tu przedstawię, to matematyczna notacja zaproponowana w latach 1980-tych na potrzeby tworzenia formalnych opisów języków programowania. To co charakterystyczne dla tych opisów, to obecność formuł przekraczających rozmiarem wszystko to, co znamy z „tradycyjnej” matematyki. Stąd też pewne konwencje notacyjne odbiegające od tradycyjnych, które jednak okazały się użyteczne. W szczególności w opisach modeli języków programowania (począwszy od rozdziału 4) zamiast symboli składających się z pojedynczych liter takich jak *a*, *b*, *c*, ... będę używał symboli wieloliterowych takich jak *sta* (dla „stan”), *den* (dla „denotacja”) itp. Dla odróżnienia formuł pisanych w **MetaSoft** od pozostałego tekstu

¹⁴ Metajęzykiem nazywa się w matematyce język służący do opisu innego języka. W naszym przypadku metajęzyk **MetaSoft** jest uniwersalnym językiem, w którym będziemy opisywali języki programowania.

książki, te pierwsze składałem czcionką Arial. Na końcu książki zamieściłem wykaz najczęściej używanych oznaczeń.

2.1.1 Notacja ogólnie-matematyczna

Na oznaczenie spójników zdaniotwórczych używam typowych słów kluczowych: **lub**, **oraz**, **nie**. Przez pp i ff oznaczam stałe logiczne „prawda” i „fałsz”. Na oznaczenie kwantyfikatorów używam symboli:

$\forall$ — *kwantyfikator ogólny* (dla każdego)

$\exists$ — *kwantyfikator egzystencjalny* (istnieje)

Zamiast pisać $i = 1, 2, \dots, n$ piszę najczęściej $i = 1; n$.

Obok tradycyjnie pisanych jednoliterowych zmiennych $a, b, c, \dots$ będę też używał zmiennych wieloliterowych, np. sta na oznaczenie zmiennej przebiegającej stany, czy też den — przebiegającej denotacje. W przypadku zmiennych wieloliterowych nie stosuję indeksów dolnych, np. sta_1 , ale piszę je na poziomie zmiennej, a więc $sta-1$. Matematykom te zasady mogą wydawać się dziwne, okażą się jednak bardzo praktyczne w opisach modeli denotacyjnych, gdzie liczba różnych zmiennych jest bardzo pokaźna.

2.1.2 Zbiory

Symbol $\emptyset$ oznacza zbiór pusty, a

$\{a_1, \dots, a_n\}$ lub $\{a_i \mid i = 1, \dots, n\}$

oznacza skończony zbiór n -elementowy. Będę również używał skrótu

$i = 1; n$ na oznaczenie $i = 1, \dots, n$.

Fakt że a należy (nie należy) do zbioru A zapisuję jako

$a : A$ ($a \not: A$)

a fakt, że jeden zbiór jest podzbiorem drugiego, jako

$A \subset B$

Przez

$A \mid B$ i $A \cap B$

oznaczam odpowiednio sumę i przecięcie zbiorów A i B . Jeżeli $\mathbf{R}$ oznacza dowolną rodzinę zbiorów, to przez

$\bigcup \mathbf{R}$

oznaczam sumę zbiorów tej rodziny. Przez

$A \times B$

oznaczam iloczyn kartezjański zbiorów. Wyrażenie bez nawiasów typu

$A \times B \times C \times D$

oznacza zbiór krotek postaci (a, b, c, d) natomiast wyrażenie

$A \times (B \times C) \times D$

oznacza zbiór krotek postaci $(a, (b, c), d)$ i analogicznie przy innych kombinacjach nawiasów. Dla każdego $n \geq 0$ n -tą potęgę kartezjańską A^{cn} zbioru A definiuję jako zbiór krotek o n elementach należących do A .

$A^{c0} = \{()\}$ — zbiór składający się z krotki pustej

$A^{cn} = \{(a_1, \dots, a_n) \mid a_i : A\}$ — dla $n > 0$

Posługując się tą operacją definiujemy kolejne dwie operacje na A :

$A^{c+} = \bigcup \{A^{ci} \mid i > 0\}$

$A^{c*} = A^{c+} \mid A^{c0}$

Zbiór wszystkich podzbiorów zbioru A i odpowiednio podzbiorów skończonych oznaczam przez:

Pod. A

SkPod. A

Będę też używał następujących oznaczeń na zbiory relacji i funkcji:

$\text{Rel.}(A, B)$ — zbiór wszystkich relacji binarnych pomiędzy A i B ; tj. zbiór wszystkich podzbiorów zbioru $A \times B$; o relacjach binarnych więcej w rozdziale 2.6,

$A \rightarrow B$ — zbiór wszystkich *funkcji częściowych* z A do B , tj. funkcji określonych dla być może nie wszystkich elementów zbioru A ,

$A \mapsto B$ — zbiór wszystkich *funkcji całkowitych* z A do B , tj. funkcji określonych dla wszystkich elementów zbioru A ; zauważmy, że każda funkcja całkowita jest częściowa, ale nie na odwrót,

$A \Rightarrow B$ — zbiór wszystkich *funkcji skończonych* z A do B , tj. funkcji określonych dla jedynie skończonej liczby elementów zbioru A ; szczególnym przypadkiem funkcji skończonej jest funkcja pusta; funkcje skończone historycznie zwane są też *mappingami*.

Zgodnie z oznaczeniami dla zbiorów zapis

$f : A \mapsto B$

oznacza, że f jest elementem zbioru $A \mapsto B$, a więc funkcją całkowitą określoną na elementach zbioru A i o wartościach w zbiorze B . I oczywiście analogicznie dla pozostałych operatorów tworzących zbiory funkcji¹⁵.

2.1.3 Funkcje

Wartości funkcji będę najczęściej zapisywał nie jak w tradycyjnej notacji matematycznej $f(a)$, ale jako $f.a$, co ma swoje względy praktyczne, o czym za chwilę. Zapis

$f.a = ?$ (2.1-1)

¹⁵ Ten przypadek wyjaśnia, dlaczego fakt należenia elementu a do zbioru A zapisujemy wyrażeniem $a:A$ zamiast — jak to się zwykło pisać w tekstach matematycznych — formułą $a \in A$.

oznacza, że funkcja f dla argumentu a nie jest określona. Nie znaczy to jednak, że „?” traktujemy jako jakąś „wartość nieokreśloną”. Zapis „ $f.a = ?$ ” jest jedynie skrótem zapisu

$$\text{nie } (\exists b)(f.a=b)$$

Podobnie skrót

$$f.a = !$$

oznacza, że funkcja f jest dla argumentu a określona, czyli

$$(\exists b)(f.a=b)$$

Dla dowolnej funkcji częściowej

$$f : A \rightarrow B$$

i dowolnego zbioru C , przez *ograniczenie funkcji f do dziedziny C* nazwę funkcję f ogr C zdefiniowaną w następujący sposób:

$$f \text{ ogr } C = \{(a, f.a) \mid a : A \cap C\}$$

Oczywiście, jeżeli A i C są rozłączne, to ograniczenie f do C jest puste. *Dziedziną określoności funkcji f* nazwę zbiór wszystkich elementów zbioru A , dla których funkcja jest określona i który oznaczę

$$\text{dom}.f = \{a \mid a : A \text{ oraz } f.a = !\}$$

Szczególnym przypadkiem ograniczenia funkcji, które przydaje się w modelach języków programowania jest usunięcie z dziedziny funkcji jednego elementu:

$$f [a/?] = f \text{ ogr } (\text{dom}.f - \{a\})$$

W opisywanych w dalszej części książki modelach języków programowania będę się często posługiwał funkcjami o wielu argumentach, dla których będę najczęściej stosował notację pochodzącą od brytyjskiego matematyka Haskell’a Curry’ego, który funkcję wielu argumentów traktuje jako funkcję, która bierze swoje argumenty niejako „jeden za drugim”, a nie „wszystkie na raz”, a więc jako funkcję

$$f : A \rightarrow (B \rightarrow (C \rightarrow (D \rightarrow E))) \quad (2.1-2)$$

Wartość takiej funkcji dla kolejnych argumentów a , b , c i d formalnie należałoby zapisać formułą

$$((((f.a).b).c).d)$$

Curry stosował jednak zapis uproszczony:

$$f.a.b.c.d$$

Mnemotechnicznie czytamy go tak:

1. funkcja f bierze element a jako argument i oddaje jako wartość funkcję $f.a$ należącą do zbioru $B \rightarrow (C \rightarrow (D \rightarrow E))$, a następnie
2. funkcja $f.a$ bierze jako argument element b i oddaje jako wartość funkcję $f.a.b$ należącą do zbioru $C \rightarrow (D \rightarrow E)$, a następnie
3. funkcja $f.a.b$ bierze jako argument element c i oddaje jako wartość funkcję $f.a.b.c$ należącą do zbioru $D \rightarrow E$, a następnie
4. funkcja $f.a.b.c$ bierze jako argument element d i oddaje jako wartość obiekt $f.a.b.c.d$ należący do zbioru E .

Notacja Curry'ego ma tę zaletę, że w złożonych wyrażeniach unika się wielu nawiasów, a ponadto można definiować funkcje o różnych typach „funkcji pośrednich” np.

$$f : A \rightarrow (B \mapsto (C \rightarrow (D \Rightarrow E))) \quad \text{lub też} \quad (2.1-3)$$

$$f : (A \rightarrow B) \mapsto (C \rightarrow (D \Rightarrow E))$$

Kolejne uproszczenie, to konwencja, że zamiast

$$f : A \rightarrow (B \mapsto (C \rightarrow (D \Rightarrow E))) \quad (2.1-4)$$

piszemy

$$f : A \rightarrow B \mapsto C \rightarrow D \Rightarrow E \quad (2.1-5)$$

Zapis

$$f : \mapsto A \quad (2.1-6)$$

oznacza, że f jest funkcją zeroargumentową o wartości należącej do zbioru A . Tę jedyną wartość oznaczamy jako $f()$.

O formułach od (2.1-2) do (2.1-6) powiemy, że opisują typy odpowiadających im funkcji. O funkcji f występującej w formule (2.1-5) powiemy więc, że jest typu:

$$A \rightarrow (B \mapsto (C \rightarrow (D \Rightarrow E)))$$

lub też, w uproszczonej postaci, typu:

$$A \rightarrow B \mapsto C \rightarrow D \Rightarrow E$$

Formalnie rzecz biorąc typem funkcji nie jest zbiór funkcji, do którego ona należy, ale napis opisujący ten zbiór, jednakże tą subtelnością, skądinąd ważną, nie będę się na razie zajmować. Jeżeli

$$f : A \mapsto A,$$

to n -tą iteracją funkcji f , gdzie $n = 0, 1, 2, \dots$, nazywam funkcję

$$f^n : A \mapsto A$$

zdefiniowaną w sposób następujący:

f^0 jest funkcją identycznościową na A , tj. $f.a = a$ dla każdego $a : A$,

$f^n.a = f.(f^{n-1}.a)$ dla $n > 0$.

W matematycznych definicjach języków programowania bardzo często stosuje się wielostopniowe definicje warunkowe funkcji zapisywane w następującej postaci:

$$\begin{aligned} f.x = & \\ & p_1.x \rightarrow g_1.x \\ & p_2.x \rightarrow g_2.x \\ & \dots \\ & p_n.x \rightarrow g_n.x \end{aligned} \quad (2.1-7)$$

gdzie p_i są klasycznymi predykatami, tj. funkcjami całkowitymi przyjmującymi wartości boolowskie pp lub ff, a g_i są funkcjami. Definicję (2.1-8) czytamy w sposób następujący:

jeżeli jest spełniony warunek $p_1.x$ to $f.x = g_1.x$, a w przeciwnym przypadku,

jeżeli jest spełniony warunek $p_2.x$ to $f.x = g_2.x$, a w przeciwnym przypadku,

... itd.

Intuicyjnie mówiąc, „wykonywanie” tej funkcji odbywa się wiersz za wierszem od góry do dołu i „zatrzymuje” się na pierwszym wierszu, w którym $p_i.x$ jest spełnione. Aby taka definicja wyznaczała funkcję w sposób jednoznaczny, dla każdego x alternatywa warunków od $p_1.x$ do $p_n.x$ powinna mieć wartość **pp**, co oznacza, że te warunki powinny wyczerpywać wszystkie możliwości. Często więc, zamiast ostatniego warunku $p_n.x$ piszemy słowo kluczowe **prawda**, co czyta się jako „a w każdym innym przypadku”. Ten zabieg pozwala uniknąć pisania złożonych warunków, a także sam przez się zapewnia wyczerpanie wszystkich możliwych sytuacji.

W schemacie (2.1-7) dopuszczam również sytuację, gdy w miejsce któregoś z $g_i.x$ stoi znak nieokreśloności ‘?’, co oznacza, że dla x spełniających $p_i.x$ funkcja f jest nieokreślona. Ta konwencja pozwana na definiowanie klauzulami warunkowymi funkcji częściowych.

W warunkowych definicjach funkcji dopuszcza się też zabieg przypominający definiowanie stałych lokalnych w programach. Na przykład, definiując funkcję $f : A \times B \mapsto C$ możemy wprowadzić następującą konstrukcję

```
f.x =
  p1.x    → g1.x
  niech
    (a, b) = x
  p2.a    → g2.x
  p3.b    → g3.x.
```

Co czytamy: *niech x będzie parą postaci (a, b)* . Możemy też użyć **niech** w inny sposób:

```
f.x =
  p1.x    → g1.x
  niech
    y = h.x
  p2.x    → g2.y
  p3.x    → g3.y.
```

Co czytamy: *niech y będzie wartością wyrażenia $h.x$* . Te wyjaśnienia nie są być może do końca formalne, staną się jednak jasne, gdy zaczniemy je stosować przy opisywaniu modeli denotacyjnych.

Funkcję skończoną całkowitą $f : \{a_1, \dots, a_n\} \mapsto \{b_1, \dots, b_n\}$, a więc mapping zdefiniowany formułą:

```
f.x =
  x=a1    → b1
  x=a2    → b2
  ...
  x=an    → bn
```

będę zapisywał w postaci

$[a_1/b_1, \dots, a_n/b_n]$ lub w postaci $[a_i/b_i \mid i = 1;n]$

a funkcję pustą jako $[]$. Niech $f : A \rightarrow B$ oraz $g : C \rightarrow D$. *Przesłonięciem funkcji f przez funkcję g* nazywamy funkcję oznaczaną przez $f \blacklozenge g : A|C \rightarrow B|D$ i zdefiniowaną w sposób następujący:

$$(f \blacklozenge g).x =$$

$$g.x = ! \rightarrow g.x$$

$$\text{prawda} \rightarrow f.x$$

Oznacza to w szczególności, że jeżeli $f.x=?$ i $g.x=?$, to $f \blacklozenge g.x=?$. Szczególnym przypadkiem przesłonięcia jest *aktualizacja funkcji* dla skończonej liczby różniących się między sobą argumentów, co zapisuję jako:

$$(f[a_1/b_1, \dots, a_n/b_n]).x =$$

$$x = a_1 \rightarrow b_1$$

...

$$x = a_n \rightarrow b_n$$

$$\text{prawda} \rightarrow f.x$$

2.1.4 Krotki

Wyrażenie

$$(a_1, \dots, a_n) \text{ lub alternatywnie } (a_i \mid i=1;n)$$

oznacza n -elementową *krotkę*¹⁶. Konsekwentnie *krotkę pustą* oznaczam przez $()$. Różnica pomiędzy wyrażeniami opisującymi zbiory, a wyrażeniami opisujące krotki jest taka, że w wyrażeniu określającym zbiór, np. $\{a, b, c, d\}$, kolejność elementów jest nieistotna i powtórzenia się „nie liczą”, a więc np.:

$$\{a, b, c, d\} = \{a, b, a, d, c\}$$

natomiast w wyrażeniu odpowiadającym krotce kolejność jest istotna, a powtórzenia możliwe, a więc np.:

$$(a, c, b, c, d) \neq (a, c, c, d, b)$$

gdzie $a, \dots, d$ są między sobą różne.

Krotki stanowią m.in. matematyczny model dla używanego w informatyce pojęcia stosu, będą więc miały do nich zastosowanie cztery funkcje, które tradycyjnie znane są pod nazwami angielskimi *push*, *pop* i *top*, którym jednak nadam nazwy polskie, bo angielskich użyję na poziomie modelu denotacyjnego przyszłego języka **Lingua**:

$$\text{kładź}((a_1, \dots, a_n), b) = (a_1, \dots, a_n, b) \quad \text{gdzie } n \geq 0$$

$$\text{usuń}.(a_1, \dots, a_n) = (a_1, \dots, a_{n-1}) \quad \text{gdzie } n \geq 2$$

$$\text{usuń}.(a) = ()$$

$$\text{usuń}(). = ()$$

$$\text{szczyt}.(a_1, \dots, a_n) = a_n \quad \text{gdzie } n \geq 1$$

¹⁶ Słowo „krotka” zostało wprowadzone do polskiej terminologii matematycznej na przełomie lat 1960/70 i pochodzi od wcześniej używanego słowa „krotność”, a to od słowa „wielokrotność”. Początkowo mówiono o parach „(a,b)”, trójkach (a,b,c) itd. i ogólnie o n -tkach $(a_1, \dots, a_n)$, by później wprowadzić ładnie brzmiące słowo „krotka” na określenie dowolnego skończonego ciągu.

szczyt.() = ?

Krotki stanowią też matematyczny model dla słów. W takim przypadku stosuje się do nich operacja konkatenacji (o której więcej w rozdz.2.4) i dwa predykaty dotyczące istnienia powtórzeń:

$$(a_1, \dots, a_n) \odot (b_1, \dots, b_m) = (a_1, \dots, a_n, b_1, \dots, b_m)$$

$$\text{są-powtórzenia.}(a_1, \dots, a_n) = \text{pp jeżeli istnieją } i \neq j \text{ takie że } a_i = a_j$$

$$\text{brak-powtórzeń.}(a_1, \dots, a_n) = \text{pp jeżeli nie istnieją } i \neq j \text{ takie że } a_i = a_j$$

Krotki można też traktować jako funkcje z liczb naturalnych w elementy krotki, co pozwala nam napisać:

$$(a_1, \dots, a_n).i = a_i.$$

Niech teraz dla pewnego zbioru A

$$\text{Krotka} = A^{c^*}$$

będzie zbiorem wszystkich krotek na zbiorze A. Na tak określonym zbiorze krotek zdefiniujemy operacje, którymi będziemy się posługiwali w przyszłości. Pierwsza z nich usuwa z krotki powtórzenia:

usuń-powtórzenia : Krotka $\mapsto$ Krotka

$$\text{usuń-powtórzenia.}(a_1, \dots, a_n) =$$

$$n = 0 \quad \rightarrow ()$$

$$n = 1 \quad \rightarrow (a_1)$$

$$n \geq 2 \quad \rightarrow$$

$$a_1 : \{a_2, \dots, a_n\} \quad \rightarrow \text{usuń-powtórzenia.}(a_2, \dots, a_n)$$

$$\text{prawda} \quad \rightarrow (a_1) \odot \text{usuń-powtórzenia.}(a_2, \dots, a_n)$$

Druga łączy dwie krotki jedna za drugą i z tak utworzonej krotki usuwa powtórzenia. Można powiedzieć, że jest to pewien odpowiednik sumy teoriomnościowej.

łącz-bez-powtórzeń : Krotka x Krotka $\mapsto$ Krotka

$$\text{łącz-bez-powtórzeń.}(k_1, k_2) = \text{usuń-powtórzenia.}(k_1 \odot k_2)$$

Trzecia operacja tworzy krotkę zawierającą wszystkie (i jedynie) elementy należące do dwóch zadanych krotek.

część-wspólna : Krotka x Krotka $\mapsto$ Krotka

$$\text{część-wspólna.}((a_1, \dots, a_n), (b_1, \dots, b_m)) =$$

$$n = 0 \quad \rightarrow ()$$

$$m = 0 \quad \rightarrow ()$$

$$a_1 : \{b_1, \dots, b_m\} \quad \rightarrow (a_1) \odot \text{część-wspólna.}((a_2, \dots, a_n), (b_1, \dots, b_m))$$

$$\text{prawda} \quad \rightarrow \text{część-wspólna.}((a_2, \dots, a_n), (b_1, \dots, b_m))$$

Czwarta operacja usuwa z zadanej krotki wszystkie elementy należące do innej zadanej krotki.

różnica : Krotka x Krotka $\mapsto$ Krotka

$$\text{różnica.}((a_1, \dots, a_n), (b_1, \dots, b_m)) =$$

$n = 0$	$\rightarrow ()$
$m = 0$	$\rightarrow (a_1, \dots, a_n)$
$a_1 : \{b_1, \dots, b_m\}$	$\rightarrow \text{różnica.}((a_2, \dots, a_n), (b_1, \dots, b_m))$
prawda	$\rightarrow (a_1) \odot \text{różnica.}((a_2, \dots, a_n), (b_1, \dots, b_m))$

Ostatnia operacja odpowiada selekcji tych elementów krotki, które spełniają pewien predykat. Niech więc

$$p : A \mapsto \{pp, ff, bb\}$$

będzie trójwartościowym predykatem określonym na zbiorze A . Każdemu takiemu predykatowi można przypisać funkcję filtrowania, która usuwa z krotki elementy niespełniające predykatu p , tj. takie dla których $p.a : \{ff, bb\}$.

$$\text{filtruj}.p : \text{Krotka} \mapsto \text{Krotka}$$

$$\text{filtruj}.p.(a_1, \dots, a_n) =$$

$$n = 0 \quad \rightarrow ()$$

$$p.a_1 = pp \quad \rightarrow (a_1) \odot \text{filtruj}.p.(a_2, \dots, a_n)$$

$$\text{prawda} \quad \rightarrow \text{filtruj}.p.(a_2, \dots, a_n)$$

2.2 Zbiory częściowo uporządkowane

Niech A będzie dowolnym zbiorem, a relacja

$$\sqsubseteq : \text{Rel}(A, A)$$

relacją binarną w tym zbiorze. O relacji $\sqsubseteq$ powiemy, że jest *częściowym porządkiem* w zbiorze A , jeżeli dla każdych $a, b, c : A$ spełnione są następujące warunki:

1. $a \sqsubseteq a$ *zwrotność*
2. jeżeli $a \sqsubseteq b$ oraz $b \sqsubseteq c$ to $a \sqsubseteq c$ *przechodność*
3. jeżeli $a \sqsubseteq b$ oraz $b \sqsubseteq a$ to $a = b$ *słaba antysymetria*

Jeżeli są spełnione jedynie dwa pierwsze warunki, to o relacji $\sqsubseteq$ powiemy, że jest *quasiporządkiem*. W dalszym ciągu będziemy mieli najczęściej do czynienia z porządkami częściowymi.

Jeżeli $a \sqsubseteq b$, to mówimy, że a jest *mniej od* b lub że b jest *więcej od* a . Jeżeli $a \sqsubseteq b$ oraz $a \neq b$, to powiemy, że a jest *istotnie mniej od* b , a b jest *istotnie więcej od* a .

Parę $(A, \sqsubseteq)$ nazwiemy *zbiorem częściowo uporządkowanym (ZCU)*. Słowo „częściowo” oznacza, że nie każde dwa elementy zbioru muszą być ze sobą porównywalne. Jeżeli jednak

dla każdych a i b albo $a \sqsubseteq b$ albo $b \sqsubseteq a$,

to o porządku mówimy, że jest *całkowity* lub *liniowy*.

Oczywiście każdy porządek całkowity jest szczególnym przypadkiem porządku częściowego, a każdy częściowy — szczególnym przypadkiem quasiporządku. Przykładem porządku częściowego, który nie jest całkowity, jest teoriomnogościowe zawieranie się zbiorów. Taki ZCU nazywamy *teoriomnogościowym ZCU*.

Niech B będzie podzbiorem częściowo uporządkowanego zbioru A i niech $b : B$.

- b nazwiemy *elementem minimalnym* w zbiorze B , jeżeli nie istnieje element $a : B$ istotnie od niego mniejszy,
- b nazwiemy *elementem najmniejszym* zbioru B , jeżeli dla każdego elementu $a : B$ zachodzi $b \sqsubseteq a$,
- b nazwiemy *elementem maksymalnym* w zbiorze B , jeżeli nie istnieje element $a : B$ istotnie od niego większy, tj. taki że $b \sqsubseteq a$ oraz $b \neq a$,
- b nazwiemy *elementem największym* zbioru B , jeżeli dla każdego elementu $a : B$ zachodzi $a \sqsubseteq b$.

Nie w każdym zbiorze częściowo uporządkowanym istnieje element minimalny, a także może ich być więcej niż jeden. Jeżeli jednak w zbiorze istnieje element najmniejszy, to jest on jedynym elementem minimalnym w tym zbiorze. Analogiczne tezy są prawdziwe dla elementów maksymalnego i największego.

Ograniczeniem górnym zbioru B nazwiemy każdy taki element zbioru A , który jest większy od wszystkich elementów zbioru B . Zauważmy, że ograniczenie górne zbioru nie musi do niego należeć, jeżeli jednak należy, to jest elementem największym tego zbioru.

Jeżeli zbiór wszystkich ograniczeń górnych zbioru B ma element najmniejszy, to ten element nazywamy *kresem górnym* zbioru B . Jeżeli zbiór dwuelementowy $\{a, b\}$ ma kres górny¹⁷, to oznaczamy go przez:

$$a \vee b$$

W teoriomnogościowym ZCU wszystkich podzbiorów jakiegoś zbioru kres górny rodziny zbiorów jest ich teoriomnogościową sumą.

Porządek częściowy $\sqsubseteq$ w zbiorze A nazwiemy *porządkiem dobrze ufundowanym*, gdy każdy niepusty podzbiór A ma element najmniejszy. Porządek, który jest liniowy i dobrze ufundowany, nazywamy *dobrym porządkiem*. Parę $(A, \sqsubseteq)$ nazywamy odpowiednio *zbiorem dobrze uporządkowanym*.

Zbiór wszystkich podzbiorów dowolnego zbioru jest częściowo uporządkowany relacją $\subset$ zawierania się zbiorów. Nie jest to jednak porządek liniowy, gdyż nie każde dwa zbiory są ze sobą porównywalne. Zbiór wszystkich liczb rzeczywistych jest liniowo uporządkowany relacją mniejszości liczb $<$, nie jest to jednak porządek dobrze ufundowany, gdyż nie każdy zbiór liczb rzeczywistych ma element najmniejszy. Zbiór liczb całkowitych dodatnich uporządkowany tą relacją jest dobrze ufundowany i jest też dobrym porządkiem. Innym przykładem porządku dobrze ufundowanego jest porządek leksykograficzny w zbiorze słów nad dowolnym skończonym alfabetem.

2.3 Zbiory łańcuchowo zupełne

Niech $(A, \sqsubseteq)$ będzie zbiorem częściowo uporządkowanym. *Łańcuchem* w tym zbiorze, nazywamy taki ciąg elementów zbioru A

$$a_1, a_2, a_3, \dots$$

że $a_i \sqsubseteq a_{i+1}$ dla $i = 1, 2, 3, \dots$. Jeżeli zbiór wszystkich elementów łańcucha ma kres górny, to nazywamy go *granicą łańcucha* i oznaczamy

¹⁷ Analogicznie definiuje się też pojęcie kresu dolnego, ponieważ jednak nie będę się nim posługiwał, nie wprowadzam tu jego definicji.

$$\lim(a_i \mid i = 1, 2, \dots)$$

O ZCU $(A, \sqsubseteq)$ powiemy, że jest *zbiorem łańcuchowo zupełnym* (ZŁZ), jeżeli każdy łańcuch ma w tym zbiorze granicę, a ponadto istnieje w A element najmniejszy. Ten najmniejszy element oznaczamy przez Φ i nazywamy *elementem dolnym* zbioru $(A, \sqsubseteq)$.

O funkcji całkowitej $f : A \mapsto A$ w zbiorze łańcuchowo zupełnym $(A, \sqsubseteq)$ powiemy, że jest *funkcją monotoniczną*, jeżeli

dla każdych a i b nierówność $a \sqsubseteq b$ implikuje nierówność $f.a \sqsubseteq f.b$.

O funkcji całkowitej $f : A \mapsto A$ w zbiorze łańcuchowo zupełnym $(A, \sqsubseteq)$ powiemy, że jest *funkcją ciągłą*, jeżeli są spełnione dwa następujące warunki:

1. dla każdego łańcucha $(a_i \mid i = 1, 2, \dots)$ ciąg $(f.a_i \mid i = 1, 2, \dots)$ tworzy również łańcuch,
 2. jeżeli pierwszy z tych łańcuchów ma kres górny, to ma go również drugi oraz
- $$\lim(f.a_i \mid i = 1, 2, \dots) = f.[\lim(a_i \mid i = 1, 2, \dots)].$$

czyli granica obrazu łańcucha jest obrazem jego granicy.

Jak łatwo zauważyć, każda funkcja ciągła jest monotoniczna, co wynika w prosty sposób z faktu, że jeżeli $a \sqsubseteq b$ to $(a, b, b, b, \dots)$ tworzy łańcuch o kresie górnym równym b .

Dla funkcji ciągłych spełnione jest bardzo ważne twierdzenie pochodzące od amerykańskiego matematyka S.C. Kleene [49]:

Twierdzenie 2.3-1 *Jeżeli f jest funkcją ciągłą w zbiorze łańcuchowo zupełnym, to zbiór rozwiązań równania*

$$x = f.x \tag{2.3-1}$$

jest niepusty i ma element najmniejszy, który wyraża się wzorem

$$Y.f = \lim(f^n.\Phi \mid n = 0, 1, 2, \dots) \blacksquare$$

Dowód tego twierdzenia jest bardzo prosty:

$$f.(Y.f) = f.(\lim(f^n.\Phi \mid n = 0, 1, 2, \dots)) = \lim(f^{n+1}.\Phi \mid n = 0, 1, 2, \dots) = \lim(f^n.\Phi \mid n = 0, 1, 2, \dots).$$

Ostatnia równość wynika z faktu, że $f^0.\Phi = \Phi$, a więc dodanie tego elementu do łańcucha nie zmienia jego granicy. ■

Występujący w dowodzie element $Y.f$ nazywamy *najmniejszym punktem stałym funkcji f* . Jest to oczywiście najmniejsze rozwiązanie równania (2.3-1), jednak w skrócie nazywamy je po prostu *rozwiązaniem*, gdyż innymi nie będziemy się zajmowali.

Pojęcie jednoargumentowej funkcji ciągłej możemy rozszerzyć na funkcje wieloargumentowe. O funkcji

$$f : A \times A \mapsto A$$

mówimy, że jest *ciągła ze względu na pierwszy argument*, gdy dla dowolnego ustalonego $b : A$, funkcja $g.a = f.(a, b)$ jest ciągła. Analogicznie definiujemy ciągłość funkcji wieloargumentowej

$$f : A \times \dots \times A \mapsto A \tag{2.3-2}$$

ze względu na jej dowolny argument. Ogólnie, o funkcji wieloargumentowej typu 2.3-2 powiemy, że jest ciągła, jeżeli jest ciągła ze względu na każdy ze swoich argumentów.

Jak się wkrótce przekonamy, funkcje ciągłe mają podstawowe znaczenie dla matematycznych modeli języków programowania, gdyż dzięki twierdzeniu Kleene'ego mogą być

wykorzystane do budowania rekurencyjnych definicji zbiorów i funkcji. Takie definicje będą z reguły przyjmować formę układów równań postaci

$$x_1 = f_1(x_1, \dots, x_n)$$

...

$$x_n = f_n(x_1, \dots, x_n)$$

Każdy taki układ równań można też potraktować jako jedno równanie w zbiorze łańcuchowo zupełnym nad iloczynem kartezjańskim $A_1 \times \dots \times A_n$:

$$X = f.X$$

gdzie

$$f.(x_1, \dots, x_n) = (f_1(x_1, \dots, x_n), \dots, f_n(x_1, \dots, x_n))$$

i gdzie porządek częściowy jest zdefiniowany „po współrzędnych”:

$$(a_1, \dots, a_n) \sqsubseteq^n (b_1, \dots, b_n) \text{ gdy } a_i \sqsubseteq b_i \text{ dla } i = 1, \dots, n.$$

Jak łatwo pokazać, jeżeli wszystkie A_i są łańcuchowo zupełne, to również ich iloczyn kartezjański z tak zdefiniowanym porządkiem jest łańcuchowo zupełny. Ponadto, jeżeli wszystkie funkcje f_i są ciągłe, to również funkcja f jest ciągła.

Okazuje się, że w układach równań stałopunktowych o funkcjach ciągłych można redukować liczbę równań i liczbę zmiennych, w podobny sposób, jak to czynimy w zwykłych równaniach algebraicznych. Przyjrzyjmy się następującemu przykładowi tych ostatnich:

$$x = 2x - y + 7$$

$$y = 3x + y - 1.$$

Ten układ można przekształcić na równoważny mu:

$$x = y - 7$$

$$y = 3x + y - 1$$

i dalej na

$$x = y - 7$$

$$y = 22/3$$

itd.

W podobny sposób możemy postępować z równaniami stałopunktowymi, co niejednokrotnie pozwala na zredukowanie ich liczby i/lub zmniejszenie ich złożoności. W takich przypadkach opieramy się na dwóch twierdzeniach udowodnionych przez Hansa Bekić'a [9] i Jacka Leszczyńskiego [52]. Dla ich sformułowania wprowadzę kolejne pojęcie.

Dwa układy równań stałopunktowych nazwiemy *równoważnymi*, jeżeli ich najmniejsze rozwiązania są identyczne.

Twierdzenie 2.3-2 *Jeżeli $f, g : A \times A \mapsto A$ są funkcjami ciągłymi, to układ równań*

$$a = f(a, b)$$

$$b = g(a, b)$$

jest równoważny układowi równań

$$a = f.(a,b)$$

$$b = g.(f.(a,b),b) \quad \blacksquare$$

Twierdzenie 2.3-3 *Jeżeli $f, g : A \times A \mapsto A$ są funkcjami ciągłymi, to układ równań*

$$a = f.(a,b)$$

$$b = g.(a,b)$$

jest równoważny układowi równań

$$a = h.b$$

$$b = g.(a,b)$$

gdzie h jest funkcją, która każdemu b przypisuje najmniejszy punkt stały jednoargumentowej funkcji $f.(x,b)$ traktowanej jako funkcja argumentu x przebiegającego zbiór A .. ■

Teoria równań stałopunktowych w zbiorach częściowo uporządkowanych stanowi ważne narzędzie pisania rekurencyjnych definicji zbiorów i funkcji występujących w denotacyjnych modelach języków programowania. Przykłady takich definicji zobaczymy w dalszych rozdziałach.

Na początku dekady lat 1970 opisałem pewną ogólną teorię równań stałopunktowych w strukturach zwanych siatkami [16]. Moich kolegów informatyków informuję, że siatka to krata zupełna — a więc również ZŁZ, ale z spełniającą mocniejsze założenia — zawierająca operację półgrupową, która jest ciągła w sensie zdefiniowanym powyżej. Siatki stanowią modele dla niżej opisanych ZŁZ języków formalnych i relacji, ale ponieważ nie korzystam z ich dodatkowych własności w mojej książce, pominąłem też ich opis.

2.4 ZŁZ języków formalnych

Na znane nam gramatyki *języków naturalnych* zwanych też *etnicznymi*, czyli języków, którymi posługują się ludzie przy komunikacji pomiędzy sobą (polski, angielski, francuski,...), można spojrzeć jako na algorytmy służące do odróżniania zdań składniowo poprawnych w danym języku, od zdań niepoprawnych czyli niegramatycznych. W tym właśnie duchu został stworzony jeden z pierwszych matematycznych modeli gramatyk języków naturalnych, a jego autorem był na początku drugiej połowy XX wieku amerykański lingwista Noam Chomsky. Opisane przez niego gramatyki są dziś znane pod nazwą *generacyjnych gramatyk bezkontekstowych* lub po prostu *gramatyk bezkontekstowych* (por. [32], [33], [34] i [35]).

Jak się dość szybko okazało, do opisów języków naturalnych ten model był zbyt ubogi, znalazł natomiast zastosowanie przy opisie języków programowania — na początku Algol 60 i Pascal, a później też ADA, CHILL i wiele innych — co spowodowało szybki rozwój teorii gramatyk bezkontekstowych. Pierwsze jej monograficzne opracowanie pochodzi z roku 1966 od S. Ginsburga [42], a pierwsze opracowanie w języku polskim mojego autorstwa [13] ukazało się w roku 1971. Rok później opublikowałem pracę [15] na temat gramatyk równaniowych równoważnych gramatykom bezkontekstowym.

W niniejszym rozdziale przypomnę w wielkim skrócie teorię języków formalnych i gramatyk bezkontekstowych Chomskiego. Tę ostatnią przedstawię jednak w wersji równaniowej. Zaczniemy od podstawowych notacji.

Niech A oznacza dowolny skończony zbiór symboli, który będziemy nazywać *alfabetem*. Słowem nad alfabetem A nazwiemy każdy skończony ciąg symboli (być może z

powtórzeniami) nad alfabetem A , a w tym i ciąg pusty, który oznaczamy przez ϵ . Słowa tradycyjnie zapisujemy bez przecinków pomiędzy poszczególnymi symbolami, np. $accdb$. Jeżeli x i y są słowami nad alfabetem A , to przez

$x \odot y$ lub po prostu xy , o ile tylko nie prowadzi to do nieporozumień, oznaczam połączenie tych dwóch słów, zwane też ich *konkatenacją*. Tak więc

$$abdaa \odot eaag = abdaaeaag$$

Konkatenację nazywamy też funkcję „ $\odot$ ” łączenia słów.

Każdy zbiór słów L nad alfabetem A nazywamy *językiem formalnym* lub po prostu *językiem* nad alfabetem A . $\text{Lan}(A)$ oznacza zbiór wszystkich języków formalnych nad A , a $\emptyset$ język (zbiór) pusty. Jeżeli P oraz Q są językami to ich *konkatenacją* nazwiemy język:

$$P \odot Q = \{p \odot q \mid p:P \text{ oraz } q:Q\}.$$

a więc „ $\odot$ ” oznacza nie tylko funkcję konkatenacji słów, ale też i języków. Ilekroć nie prowadzi to do nieporozumień, zamiast $P \odot Q$ piszę PQ . Jak łatwo sprawdzić, operacja konkatenacji jest łączna, uprawnione jest więc pisanie PQL w miejsce $(PQ)L$ oraz $P(QL)$. Będę zakładał, że operator konkatenacji wiąże mocniej niż suma, co oznacza, że wyrażenie

$$(P \odot Q) \mid (R \odot S)$$

można zapisać w postaci

$$PQ \mid RS$$

Jak łatwo udowodnić, konkatenacja jest rozdzielna względem sumy teoriomnogościowej, czyli:

$$(P \mid Q)R = PR \mid QR.$$

Przez *n -tą potęgę języka P* , gdzie n jest liczbą całkowitą nieujemną, nazywamy język określony równaniami:

$$P^0 = \{\epsilon\}$$

$$P^n = P \odot P^{n-1} \text{ dla } n > 0$$

Będę też używał dwóch dalszych operatorów na językach formalnych:

$$P^+ = \text{suma } P^n \text{ dla } n > 0$$

$$P^* = P^+ \mid P^0$$

Tak więc dla alfabetu A , zbiór A^+ jest zbiorem wszystkich niepustych słów nad A , a A^* — zbiorem wszystkich słów nad A . Języki nad A to po prostu podzbiory zbioru A^* .

Zbiór $(\text{Lan}(A), \subset)$ częściowo uporządkowany relacją zawierania się zbiorów $\subset$ tworzy zbiór łańcuchowo zupełny, w którym język pusty $\emptyset$ jest elementem dolnym. Jak łatwo pokazać, wszystkie wyżej zdefiniowane operacje na językach, a także teoriomnogościowa suma języków są operacjami ciągłymi w tym ZŁZ. Dla każdych dwóch języków P i Q ich najmniejsze ograniczenie górne to po prostu ich suma teoriomnogościowa $P \mid Q$, a granica łańcucha języków to suma teoriomnogościowa wszystkich jego elementów.

Czytelnika może dziwić fakt, że w rozdziale 2.1.2 wprowadziłem operację potęgi kartezjańskiej, a obecnie, bardzo do niej podobną operację potęgi na językach. W rzeczywistości jednak operacje te nie są jednakowe, gdyż opierają się na dwóch różnych operacjach podstawowych: iloczynu kartezjańskiego i konkatenacji. Zauważmy, że jeżeli P i Q są językami, to

$$P \odot Q \text{ jest językiem składającym się ze słów postaci } p \odot q, \text{ gdzie } p:P \text{ i } q:Q,$$

natomiast

$P \times Q$ jest zbiorem par słów postaci (p, q) , gdzie $p : P$ i $q : Q$.

Konkatenacja dwóch języków jest więc nadal językiem, podczas gdy ich iloczyn kartezjański już nie jest.

2.5 Gramatyki równaniowe

Ponieważ wszystkie operacje na językach opisane w rozdziale 2.4 są ciągłe, można ich używać do definiowania języków za pomocą równań stałopunktowych w sensie opisanym w rozdziale 2.3.

Rozpocznijmy od przykładu prostego układu równań definiującego język identyfikatorów jakiegoś hipotetycznego języka programowania (zakładamy, że identyfikatory zaczynają się od znaku litery):

Litera = $\{a, b, \dots, z\}$

Cyfra = $\{0, 1, \dots, 9\}$

Znak = Litera | Cyfra

Sufiks = $\{\epsilon\}$ | Znak © Sufiks

Identyfikator = Litera © Sufiks

Takie układy równań nazywam *gramatykami równaniowymi*, a ich najmniejsze rozwiązania, a więc krotki języków — *wielorodzajowymi językami formalnymi*. W powyższym przypadku na definiowany język składa się pięć kategorii składniowych (języków), które właśnie nazywam *rodzajami*:

Litera, Cyfra, Znak, Sufiks, Identyfikator

Kategoria Sufiks ma charakter pomocniczy, gdyż służy jedynie do wyrażenia faktu, że identyfikator musi rozpoczynać się od litery. Odpowiadające jej równanie można jednak wyeliminować (por. Twierdzenie 2.3-2 i Twierdzenie 2.3-3), gdyż — jak łatwo udowodnić — rozwiązaniem tego równania jest zbiór wszystkich słów nad alfabetem Znak włączywszy w to słowo puste ϵ , a więc zbiór Znak*. Można też wyeliminować równanie trzecie, ostatecznie sprowadzając nasz układ do postaci:

Litera = $\{a, b, \dots, z\}$

Cyfra = $\{0, 1, \dots, 9\}$

Identyfikator = Litera © (Litera | Cyfra)*

Ten układ to również gramatyka równaniowa, jednakże innego już języka wielorodzajowego, bo składającego się jedynie z trzech kategorii. Oczywiście te trzy kategorie pokrywają się z odpowiadającymi im trzema kategoriami wcześniej zdefiniowanego języka o pięciu kategoriach.

Zajmijmy się teraz gramatykami równaniowymi w sposób bardziej formalny, tak jak zostało to opisane w pracy [15]. Niech A będzie dowolnym skończonym niepustym alfabetem, a

$Kla \subset Lan(A)$

dowolną klasą języków nad A . Niech $Wie(Kla)$ oznacza najmniejszą klasę funkcji typu

$w : Lan(A)^{cn} \mapsto Lan(A)$

dla wszystkich $n \geq 0$ zawierającą:

- (1) wszystkie rzutowania, tj. funkcje postaci $f.(X_1, \dots, X_n) = X_i$ dla $i \leq n$,
- (2) wszystkie funkcje stałe, w tym zeroargumentowe, o wartościach w Kla ,
- (3) funkcje sumy i konkatencji języków

i zamkniętą ze względu na składanie (superpozycję) funkcji.

Funkcje należące do tej klasy nazywam *wielomianami nad klasą* Kla . Zauważmy, że ponieważ funkcje opisane w (1), (2) i (3) są ciągłe, a ciągłość funkcji jest niezmiennikiem składania, wielomiany są funkcjami ciągłymi.

Językiem atomowym nad A nazywam każdy język jednoelementowy $\{s\}$, gdzie $s : A^*$. Wielomiany zbudowane nad dowolnym zbiorem języków atomowych będę nazywał *wielomianami Chomskiego*¹⁸. Przykłady takich wielomianów to:

$$w_1.(X, Y, Z) = \{b\}$$

$$w_2.(X, Y) = \{b\}$$

$$w_3.(X, Y, Z) = X$$

$$w_4.(X, Y, Z) = (\{d\}X\{b\}YY\{c\} \mid X) Z$$

Zauważmy, że dla pełnego określenia wielomianu musimy explicite podać jego arność, co wiadać na przykładzie dwóch różnych wielomianów w_1 i w_2 .

Funkcje zdefiniowane tak jak wielomiany, ale bez użycia sumy nazywamy *jednomianami*. Wielomiany w_1 , w_2 i w_3 są właśnie jednomianami. Ze względu na rozdzielność konkatencji względem sumy, każdy wielomian da się jednoznacznie przedstawić w postaci sumy jednomianów.

Gramatyką równaniową nad alfabetem A nazwiemy każdy stałopunktowy układ równań postaci:

$$\begin{aligned} X_1 &= w_1.(X_1, \dots, X_n) \\ &\dots \\ X_n &= w_n.(X_1, \dots, X_n) \end{aligned} \tag{2.5.1}$$

gdzie po prawych stronach równości mamy wielomiany Chomskiego nad A . Ponieważ wielomiany są funkcjami ciągłymi, to każdy taki układ ma rozwiązanie $(L_1, \dots, L_n)$. O językach $L_1, \dots, L_n$ powiemy, że są *zdefiniowane gramatyką równaniową* (2.5.1), a także, że są *równaniowo definiowalne*.

Jak udowodniono w [15], klasa języków równaniowo definiowalnych jest identyczna z klasą języków bezkontekstowych w sensie Chomskiego¹⁹. Klasa języków równaniowo definiowalnych nie ulegnie też zmianie, jeżeli przy budowaniu wielomianów dopuścimy teoriojęzykowe operacje „*” i „+”, co pozwala niekiedy uprościć gramatykę równaniową. Ta klasa nie ulegnie też zmianie, jeżeli wielomiany będziemy budowali nie jedynie nad językami atomowymi, ale

¹⁸ Noam Chomsky — amerykański lingwista, filozof, działacz polityczny. Profesor lingwistyki w Massachusetts Institute of Technology, współtwórca pojęcia gramatyki transformacyjno-generatywnej — nie wprowadził tego pojęcia, jednakże wielomiany nad językami atomowymi dość blisko odpowiadają bezkontekstowym gramatykom generacyjnym Chomskiego.

¹⁹ Co oznacza, że dla każdej gramatyki równaniowej istnieje równoważna jej gramatyka bezkontekstowa i na odwrót.

nad dowolnymi językami równaniowo definiowalnymi. Dowody wszystkich tych faktów można znaleźć w [15].

Ponieważ języki równaniowo definiowalne są tożsame z lepiej znanymi z literatury językami bezkontekstowymi, w przyszłości będę się posługiwał tym właśnie pojęciem.

2.6 ZŁZ relacji binarnych

Niech A i B będą dowolnymi zbiorami. *Relacją binarną (dwuczłonową)* albo po prostu *relacją* pomiędzy tymi dwoma zbiorami, nazwiemy każdy podzbiór zbioru $A \times B$. Zbiór wszystkich relacji pomiędzy tymi zbiorami oznaczamy przez:

$$\text{Rel.}(A,B) = \{R \mid R \subset A \times B\}$$

natomiast $a R b$ będzie oznaczało $(a,b):R$. Jeżeli $A = B$, to zamiast $\text{Rel}(A,A)$ piszemy $\text{Rel}(A)$. Dla każdego zbioru A definiujemy *relację identycznościową* nad A :

$$[A] = \{(a, a) \mid a:A\}$$

Przez $\emptyset$ oznaczamy *relację pustą* (pusty zbiór par)²⁰. Jeżeli oznaczymy $\text{Boolowska} = \{\text{pp}, \text{ff}\}$, gdzie pp i ff są odpowiednio wartościami logicznymi „prawda” i „fałsz” oraz

$$\text{pre} : A \rightarrow \text{Boolowska}$$

jest predykatem (być może częściowym) na zbiorze A , to

$$\text{Id}(\text{pre}) = \{[a \mid \text{pre}.a = \text{pp}]\}$$

czyli $\text{Id}(\text{pre})$ jest relacją (funkcją) identycznościową ograniczoną do argumentów spełniających predykat pre . Jeżeli $R : \text{Rel}(A,B)$, to

$$\text{dom}.R = \{a \mid (\exists b:B) a R b\} \text{ — dziedziną relacji } R$$

$$\text{cod}.R = \{b \mid (\exists a:A) a R b\} \text{ — przeciwdziedzina } R$$

Niech $P : \text{Rel}(A,B)$ i $R : \text{Rel}(B,C)$. *Złożeniem sekwencyjnym* relacji P i R nazywamy relację $P \bullet R : \text{Rel}(A,C)$ zdefiniowaną następująco:

$$P \bullet R = \{(a, c) \mid (\exists b:B) (a P b \ \& \ b R c)\}$$

Dla każdych dwóch relacji ich złożenie zawsze istnieje, ale może być relacją pustą. Jak łatwo sprawdzić operacja $\bullet$ jest łączna, tzn.

$$(P \bullet R) \bullet Q = P \bullet (R \bullet Q)$$

Uprawnione jest więc pisanie wyrażeń typu $P \bullet R \bullet Q$. Ilekroć nie będzie to prowadziło do nieporozumień, zamiast $P \bullet R$, będę pisał PR . Będę też zakładał, że operator złożenia $\bullet$ wiąże mocniej od operatora sumy $|$, a więc wyrażenie $(P \bullet R) \mid (Q \bullet S)$ będę zapisywał jako

$$PR \mid QS$$

W dalszej części książki składanie relacji będzie stosowane w szczególnym przypadku, gdy relacje są funkcjami. W tym przypadku:

$$(P \bullet R).a = R.(P.a)$$

a stąd

²⁰ Tym samym symbolem oznaczam zbiór pusty (rozdział 2.1). Kolegów matematyków proszę o wybaczenie, a kolegom nie-matematykom wyjaśniam, że różnica pomiędzy zbiorem pustym, a pustą relacją jest taka jak pomiędzy wodą sodową bez soku malinowego, a wodą sodową bez soku truskawkowego.

$$(P \bullet R \bullet Q).a = (P \bullet (R \bullet Q)).a = Q.(R.(P.a)),$$

co oznacza, że sekwencyjnym złożeniu funkcji, składane funkcje są „wykonywanie” od lewej do prawej.

Dla każdej relacji $R: \text{Rel}(A)$ definiuję — analogicznie jak dla języków — operatory *potęgowania* oraz odpowiadające jej iteracyjne operatory „gwiazdkowania” i „plusowania”:

$$R^0 = [A] \text{ — relacja identyczności nad } A.$$

$$R^n = RR^{n-1} \text{ dla } n > 0$$

$$R^+ = \bigcup \{R^n \mid n > 0\}$$

$$R^* = R^+ \mid R^0$$

Odwrotnością relacji R nazywamy relację R^{-1} taką, że

dla każdych a i b , $a R^{-1} b$ wtedy i tylko wtedy gdy $b R a$.

O relacji R powiemy, że jest *funkcją*, gdy

dla każdych a, b i c , jeżeli $a R b$ i $a R c$, to $b = c$.

Jeżeli R i R^{-1} są funkcjami, to powiemy, że R jest *funkcją odwracalną* lub też *funkcją wzajemnie jednoznaczną*. Jeżeli R i P są funkcjami, to ich złożenie RP jest również funkcją (być może pustą) oraz

$$(RP).a = P.(R.a)$$

a więc w przypadku funkcji złożenie jest superpozycją.

Zbiór relacji $\text{Rel}(A, B)$ tworzy zbiór łańcuchowo zupełny z uporządkowaniem teoriomnogociowym i relacją pustą jako elementem dolnym. Wszystkie opisane wyżej operacje na relacjach są ciągłe. W przyszłości będziemy często korzystali z następującego twierdzenia:

Twierdzenie 2.6-1 *Dla każdych $P, Q : \text{Rel}(A)$ najmniejszymi rozwiązaniami równań stałopunktowych:*

$$X = P \mid QX \quad \text{oraz}$$

$$X = P \mid XQ$$

są odpowiednio

$$X = Q^*P \quad \text{oraz}$$

$$X = P^*Q$$

Ponadto, jeżeli P i Q są funkcjami o rozłącznych dziedzinach, to oba te rozwiązania są również funkcjami. ■

W tym miejscu warto zauważyć, że dowolny zbiór funkcji częściowych

$$A \rightarrow B$$

tworzy łańcuchowo zupełny podzbiór $(\text{Rel}(A, B), \subseteq)$, który jest zamknięty ze względu na operację złożenia „ $\bullet$ ” funkcji, a także sumy funkcji, ale o rozłącznych dziedzinach. Ponieważ obie te funkcje są ciągłe w $\text{Rel}(A, B)$ możemy budować rekurencyjne (stałopunktowe) definicje funkcji występujących często w denotacyjnych definicjach języków programowania. Zauważmy, że skoro A i B są dowolnymi zbiorami, to w szczególności definicje stałopunktowe można też budować dla dowolnych funkcji typu:

$$f : A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_n$$

Wymaga to tylko zdefiniowania odpowiednich operatorów. Dla przykładu rozważmy rekurencyjną definicję funkcji potęgowania odwołującą się do funkcji mnożenia i odejmowania, którą zapiszę w postaci „zwykłej” a nie Curry’ego²¹:

potęga : Liczba x Liczba $\rightarrow$ Liczba

potęga.(n, m) =

m = 0 $\rightarrow$ 1

m > 0 $\rightarrow$ n * potęga.(n, (m-1))

gdzie

Liczba = {0, 1, ...}

Chcemy pokazać, że ta definicja da się zapisać w postaci równania stałopunktowego w teoriomnogościowym ZŁZ relacji:

Rel.(Liczba x Liczba, Liczba) (*)

Zbuduję więc równanie stałopunktowe, którego rozwiązaniem będzie funkcja

potęga.(n, m) = n^m

traktowana jako relacja z naszego ZŁZ. Rozpocznę od zdefiniowania pewnej szczególnej operację składania dowolnych funkcji typu

F, Q : Rel.(A x A, A)

Złożeniem F i Q na drugim argumencie nazwę relację

F $\circledast$ Q = {{a,b,c} | ($\exists d$) (a,b,d) : F oraz (a,d,c) : Q}

Jeżeli F i Q są funkcjami, to:

[F $\circledast$ Q].(a,b) = Q.(a, F.(a,b))

Zbiór relacji typu (**) tworzy oczywiście zbiór łańcuchowo zupełny z teoriomnogościowym zawieraniem. Można pokazać, że operator $\circledast$ jest ciągły ze względu na pierwszy argument. Rozważmy teraz równanie

potęga = zero | (minus $\circledast$ potęga) $\circledast$ razy (**)

gdzie:

zero.(n, 0) = 1

minus.(n, m) = m-1 dla m > 0, a dla m = 0 funkcja jest nieokreślona

razy.(n, m) = n^m

Ponieważ suma teoriomnogościowa i nasze składanie są ciągłe w ZŁZ relacji typu (*), to z twierdzenia Kleene’ego wynika, że rozwiązaniem równania (**) będzie granica łańcucha relacji:

$P^0 \subset P^1 \subset P^2 \subset \dots$ (***)

które są funkcjami określonymi w następujący sposób:

²¹ W tym miejscu wprowadzam też konwencję typową dla MetaSoft i innych podobnych metajęzyków, a polegającą na używaniu mnemotechnicznych symboli wieloliterowych takich jak „potęga”, „liczba” itp. Jest to bardzo użyteczna konwencja przy opisywaniu systemów programowania, gdzie liczby wprowadzanych symboli idą w dziesiątki, a nawet setki.

$$P^0 = \text{zero}$$

$$P^{i+1} = (\text{minus } 2 \ P^i) \ 2 \text{ razy} \quad \text{dla } i \geq 0$$

Jak stąd wynika dla każdego $i \geq 0$:

$$P^{i+1}.(n, m) = P^i.(n, m-1) * m$$

co oznacza, że dla każdego $i \geq 0$ funkcja P^i jest częściową funkcją potęgowania ograniczoną do $m \leq i$:

$$P^i.(n, m) =$$

$$m \leq i \rightarrow m^i$$

prawda $\rightarrow ?$

Ponieważ te wszystkie funkcje pokrywają się na częściach wspólnych ich dziedzin, teoriomnogościową sumą łańcucha (***) jest funkcja i jest to funkcja potęgowania określona dla dowolnych $n, m \geq 0$.

2.7 ZŁZ dziedzin denotacyjnych

W denotacyjnych modelach języków programowania ważną rolę odgrywają zbiory historycznie nazywane *dziedzinami*. Te dziedziny z reguły definiujemy układami równań, a w tym równań stałopunktowych, w których korzystamy z niżej wymienionych operacji na zbiorach. Większość z nich była już wymieniana wcześniej, jednak powtarzam je tu wszystkie dla pełności obrazu:

- 1) $A \mid B$ — suma teoriomnogościowa zbiorów
- 2) $A \cap B$ — przecięcie teoriomnogościowe zbiorów
- 3) $A \times B$ — iloczyn kartezjański zbiorów
- 4) A^{cn} — kartezjańska n -ta potęga zbioru A
- 5) A^{c+} — kartezjańska $+$ -iteracja zbioru A (rozdz.2.1)
- 6) A^{c*} — kartezjańska $*$ -iteracja zbioru A (rozdz.2.1)
- 7) $\text{SkPod}.A$ — zbiór wszystkich skończonych niepustych podzbiorów zbioru A
- 8) $A \Rightarrow B$ — zbiór wszystkich funkcji skończonych z A do B (w tym funkcja pusta)
- 9) $A - B$ — różnica zbiorów
- 10) $\text{Pod}.A$ — zbiór wszystkich podzbiorów zbioru A
- 11) $A \rightarrow B$ — zbiór wszystkich funkcji z A do B
- 12) $A \mapsto B$ — zbiór wszystkich funkcji całkowitych z A do B
- 13) $\text{Rel}.(A, B)$ — zbiór wszystkich relacji z A do B

Tych wszystkich operacji będziemy używali przy definiowaniu dziedzin nie tylko w sposób prosty równaniami typu:

$$\text{Stan} = \text{Identyfikator} \Rightarrow \text{Obiekt}$$

$$\text{Instrukcja} = \text{Stan} \rightarrow \text{Stan} \quad (2.7-1)$$

ale też i równaniami istotnie stałopunktowymi jak np.:

Rekord = Atrybut $\Rightarrow$ Obiekt

Obiekt = Liczba | Rekord (2.7-2)

Definicja (2.7-1) nie budzi żadnych wątpliwości, jednakże druga ma postać układu równań stałopunktowych, powstaje więc pytanie, czy jest matematycznie legalna, tj. czy posiada jednoznacznie wyznaczone rozwiązanie. Tego typu pytania zadawałem w kontekście zbiorów łańcuchowo zupełnych (w rozdziale 2.3), gdzie ciągłość funkcji gwarantuje istnienie jej najmniejszego punktu stałego. Tu jednakże sytuacja jest nieco odmienna. Mamy co prawda porządek częściowy pomiędzy zbiorami, którym jest relacja zawierania $\subset$, jednak w jakim zbiorze ten porządek miałby być określony? Zbiór wszystkich zbiorów nie wchodzi w grę, gdyż jak wiadomo nie istnieje²². Problem jednak daje się rozwiązać. Jak pokazał M.P. Cohn [36], dla dowolnej rodziny (nie necessarily zbioru!) zbiorów B można zbudować zbiór zbiorów $Zbiór.B$ o następujących własnościach:

1. należą do niej wszystkie zbiory rodziny B ,
2. jest zamknięta ze względu na wszystkie operacje od 1) do 13),
3. jest zamknięta ze względu na sumy przeliczalnych rodzin jej elementów,
4. należy do niej zbiór pusty $\emptyset$.

A skoro tak, to jako rodzinę B przyjmujemy rodzinę zbiorów wyjściowych dla budowania dziedzin denotacyjnych danego systemu, składającą się np. ze zbioru liczb całkowitych, zbioru liczb rzeczywistych, zbioru znaków ASCII i zbioru pustego. Ponieważ $(Zbiór.B, \subset)$ tworzy zbiór łańcuchowo zupełny, można więc mówić o ciągłości określonych na nim funkcji. Jak można pokazać, zdefiniowane na początku rozdziału funkcje na zbiorach od 1) do 8) są ciągłe, różnica zbiorów jest ciągła jedynie względem pierwszego argumentu, a pozostałe operacje nie są ciągłe, a więc nie mogą występować w równaniach stałopunktowych²³.

Jak zatem widzimy równanie stałopunktowe (2.7-2) ma jednoznacznie wyznaczone (najmniejsze) rozwiązanie określone twierdzeniem Kleene'ego (rozdziale 2.3). Zdefiniowane przy jego pomocy rekordy dopuszczają dowolną głębokość zagnieżdżenia, co oznacza, że wartościami w tych rekordach mogą być rekordy, ale mniejszej głębokości, których wartościami mogą być znów rekordy, ale jeszcze mniejszej głębokości, itd. a na samym końcu są zawsze rekordy, których wartościami są liczby. Gdybyśmy jednak w tym równaniu operator $\Rightarrow$ tworzenia funkcji skończonych zastąpili operatorem $\rightarrow$ tworzenia funkcji częściowych (a więc również nieskończonych), to takie równanie nie miałoby rozwiązania. Z tym właśnie

²² Formalnie rzecz biorąc próba wprowadzenia pojęcia zbioru wszystkich zbiorów prowadzi do dobrze znanego paradoksu. Gdyby bowiem taki zbiór istniał, to można by go podzielić na dwa rozłączne podzbiory: Z_1 — zbiór wszystkich zbiorów, które są swoimi elementami i Z_2 — zbiór wszystkich zbiorów, które swoimi elementami nie są. Zbiór Z_2 , który jest oczywiście zbiorem musiałby więc należeć do jednego z nich. Nie może jednak należeć do Z_1 , bo do niego należą jedynie te, które są swoimi elementami, ale nie może też należeć do Z_2 , czyli do siebie, bo wtedy musiałby do siebie nie należeć. Wyrażając rzecz intuicyjnie: wszystkich zbiorów jest za dużo, aby tworzyły zbiór.

²³ Dla przykładu pokażmy, że operator $\rightarrow$ nie jest ciągły. Weźmy więc dowolny łańcuch zbiorów $A_1 \subset A_2 \subset \dots$ różnych między sobą i dowolny zbiór B . Ciąg dziedzin funkcji $A_i \rightarrow B$ tworzy co prawda łańcuch, jednakże do żadnego z jego elementów nie należy żadna funkcja całkowita na sumie $U A_i$, a więc nie należy też do $U(A_i \rightarrow B)$, co oznacza, że $U(A_i \rightarrow B) \neq U A_i \rightarrow B$. W analogiczny sposób można pokazać, że operator nie jest ciągły względem drugiego argumentu, a także że nieciągłe są operatory $A \mapsto B$ oraz $Rel(A, B)$. Zauważmy jednak, że $U(A_i \Rightarrow B) = U A_i \Rightarrow B$ i podobnie względem prawego argumentu, a więc operator $\Rightarrow$ jest obustronnie ciągły.

problemem zderzyli się matematycy, którzy w latach 1970. próbowali zdefiniować semantykę denotacyjną języka Algol 60, o czym piszę w rozdziale 4.1.

Zakaz używania funkcji nieciągłych w układach równań stałopunktowych nie oznacza oczywiście, że nie mogą one występować w układach, które da się wydzielić w odrębnym bloku równań nie mających stałopunktowego charakteru. Na przykład, nasze dwa układy (2.7-1) i (2.7-2) można „całkiem legalnie” połączyć w jeden:

$$\begin{array}{ll} \text{Instrukcja} &= \text{Stan} \rightarrow \text{Stan} \\ \text{Stan} &= \text{Identyfikator} \Rightarrow \text{Obiekt} \\ \text{Obiekt} &= \text{Liczba} \mid \text{Rekord} \\ \text{Rekord} &= \text{Etykieta} \Rightarrow \text{Obiekt} \end{array} \quad (2.7-3)$$

Rekurencyjne względem siebie są jedynie Obiekt i Rekord, a definiujący ich układ równań stałopunktowych obejmuje jedynie funkcje ciągłe. Funkcja nieciągła „ $\rightarrow$ ” występuje w układzie, który nie ma charakteru stałopunktowego.

2.8 Błędy abstrakcyjne

Dla praktyczne wszystkich wyrażeń występujących w programach ich wartości w pewnych okolicznościach nie dają się wyliczyć. Oto kilka typowych przykładów:

- nie da się wyliczyć wartości wyrażenia x/y , jeżeli $y = 0$,
- nie da się wyliczyć wartości wyrażenia $x+1$, jeżeli zmienna x nie została w programie zadeklarowana,
- nie da się wyliczyć wartości wyrażenia $x+y$, jeżeli jego wartość wykroczyłaby poza dopuszczalny w danej implementacji zakres,
- nie da się wyliczyć wartości wyrażenia tablicowego $a[k]$, jeżeli wartość indeksu k przekracza rozmiar tablicy a ,
- nie da się wykonać kwerendy Czy Jan Kowalski jest w wieku emerytalnym? dla bazy, w której Jan Kowalski nie występuje.

W tych wszystkich sytuacjach należy oczekiwać, że interpreter zatrzyma wykonanie programu i wyświetli na monitorze komunikatu błędu.

Aby opisać ten mechanizm formalnie, wprowadza się do dziedzin elementy zwane *błędami abstrakcyjnymi*. W ogólnym przypadku mogą one być czymkolwiek, jednak w naszym przypadku przyjmuję, że są to napisy, np. ‘dzielenie-przez-zero’. Ujmuję je w apostrofy, aby odróżniały się od metazmiennych, tj. zmiennych z poziomu MetaSoftu.

Fakt, że próba dzielenia przez zero powoduje wyświetlenie błędu, zapisuję teraz w postaci równości:

$$\text{liczba} / 0 = \text{'dzielenie-przez-zero'}$$

W przypadku ogólnym dla każdej dziedziny danych $Dana$ wprowadzam odpowiadającą jej dziedzinę danych z błędami:

$$DanaB = Dana \mid \text{Błąd}$$

gdzie Błąd jest jakimś ustalonym zbiorem błędów abstrakcyjnych. Po takim zabiegu każdą częściową operację na danych

$$\text{op} : \text{Dana}_1 \times \dots \times \text{Dana}_n \rightarrow \text{DanaB}$$

trzeba „dookreślić” na błędach, tj. rozszerzyć do operacji całkowitej

$$\text{opb} : \text{DanaB}_1 \times \dots \times \text{DanaB}_n \mapsto \text{DanaB}$$

Oczywiście opb powinna pokrywać się z op na obszarze jej określoności, tzn. jeżeli $d_1, \dots, d_n$ nie są błędami oraz $\text{op}.(d_1, \dots, d_n)$ jest określone, to $\text{opb}.(d_1, \dots, d_n) = \text{op}.(d_1, \dots, d_n)$.

O operacji opb powiemy, że jest *transparentna dla błędów*, lub w skrócie *transparentna*, gdy spełnia następujący warunek:

$$\text{jeżeli } d_k \text{ jest pierwszym błędem w ciągu } d_1, \dots, d_n, \text{ to } \text{opb}.(d_1, \dots, d_n) = d_k$$

Ten warunek oznacza, że argumenty operacji są wyliczane kolejno od lewej do prawej i pierwszy napotkany błąd staje się wynikiem końcowym obliczenia.

Większość definiowanych w przyszłości operacji na danych będzie transparentna. Wyjątek stanowią operacje boolowskie, o których piszę w rozdziale 2.9.

Często mechanizm błędów jest implementowany w ten sposób, że błędy służą jedynie do poinformowania użytkownika systemu o pojawianiu się nieprawidłowości i zatrzymaniu obliczeń. O takim mechanizmie błędów powiem, że jest *reaktywny*. Może być jednak i tak, że pojawianie się błędu powoduje wykonanie pewnej specjalnej akcji obsługi błędu, np. przywróceniu zapamiętanego stanu bazy danych (patrz rozdział 12.7.6.4). Taki mechanizm obsługi błędów będę nazywał *proaktywnym*.

Jak przekonamy się w dalszym ciągu książki, model reaktywny może być w prosty sposób uzupełniony do proaktywnego. Ponieważ jednak ten drugi jest technicznie bardziej złożony, w dyskusji nad wszystkimi wersjami prototypowego języka **Lingua** poza **Lingua-SQL** przyjmę głównie wersję reaktywną, choć z pewnymi wyjątkami (rozdziały 6.1.8, 7.3.3 oraz 12.7.6.4).

Dobrze zbudowany system obsługi błędów pozwala uniknąć sytuacji — jakże irytujących dla użytkownika — gdy program zawiesza działanie informując nas jedynie, że jest błąd, ale nie wskazując miejsca, gdzie należy go szukać. Pozwala też uniknąć sytuacji znacznie gorszych, gdy zamiast sygnalizować błąd program generuje wynik niepoprawny, nie informując o tym użytkownika (por. rozdział 11.2).

2.9 Trójwartościowy rachunek zdań

Tertium non datur — mawiali starożytni. Komputery zaprzeczyły tej prawdzie.

W klasycznej logice Arystotelesa przyjmuje się, że każda teza jest albo prawdziwa, albo fałszywa. Trzeciej możliwości nie ma. Jednakże w świecie komputerów trzecia możliwość nie tylko nie jest wykluczona, ale jest nie do uniknięcia. Wiąże się to z omawianym w rozdziale 2.8 faktem, że przy wyliczaniu wartości złożonego wyrażenia boolowskiego, np. $x/y > 2$, może pojawić się błąd.

Dla opisanego mechanizmu reagowania na błędy w przypadku wyrażeń boolowskich obok dziedziny klasycznych wartości logicznych

$$\text{Boolowska} = \{\text{pp}, \text{ff}\}$$

wprowadzamy dziedzinę boolowską uzupełnioną o trzeci element

$$\text{BoolowskaB} = \{\text{pp}, \text{ff}, \text{bb}\}$$

gdzie bb reprezentuje błąd lub nieskończone obliczenie. Dla uproszczenia rozważań przyjmuję, że błąd jest jeden, co mogę uczynić bez straty na ogólności, gdyż w tym przypadku mechanizm

obsługi błędów będzie zawsze reaktywny. Okazuje się jednak, że przyjęcie dla operatorów logicznych opisaney w rozdziale 2.8 zasady transparentności nie byłoby najlepszym rozwiązaniem. Aby to pokazać rozważmy następującą instrukcję warunkową²⁴:

```
if x ≠ 0 and 1/x < 10 then x := x+1 else x := x-1 fi
```

Oczekiwalibyśmy zapewne, by dla $x=0$ wykonało się przypisanie $x:=x-1$. Gdyby jednak koniunkcja była transparentna, to wyrażenie $x \neq 0$ and $1/x < 10$ przyjęłoby wartość ‘dzielenie-przez-zero’, a więc wykonanie programu zostałoby zatrzymane.

Zauważmy też, że założenie transparentności spójnika **oraz** oznaczałoby spełnienie równości:

ff **oraz** bb = bb

co implikowałoby wymaganie, aby interpreter obliczający wartość wyrażenia **p oraz q** najpierw — jak w „zwykłej” matematyce — dokonał wyliczenia wartości obu argumentów, a dopiero później zastosował do nich operator **oraz**. Taki tryb wyliczania wartości wyrażeń nazywamy *ewaluacją ochoczą* (ang. *eager evaluation*).

Alternatywą dla niej jest *ewaluacja leniwa* (ang. *lazy evaluation*). W tym trybie, jeżeli wartość pierwszego argumentu koniunkcji jest ff, to nie liczymy wartości drugiego argumentu i przyjmujemy, że wartość wynikowa jest również ff. Analogicznie postępujemy dla **lub** gdy jego pierwszy argument jest pp. Tak więc:

ff **oraz** bb = ff

pp **lub** bb = pp

Trójwartościowy rachunek zdań realizujący powyższą zasadę leniwej ewaluacji zaproponował w roku 1961 John McCarthy [56]. Jego spójniki zdaniotwórcze są opisane Tab. 2.9-1 (przyrostek „-m” od McCarthy).

lub-m	pp	ff	bb
pp	pp	pp	pp
ff	pp	ff	bb
bb	bb	bb	bb

oraz-m	pp	ff	bb
pp	pp	ff	bb
ff	ff	ff	ff
bb	bb	bb	bb

nie-m	
pp	ff
ff	pp
bb	bb

Tab. 2.9-1 Tabelki definicyjne spójników logicznych McCarthy’ego

By wyjaśnić intuicję leżącą u podstaw tego rachunku przeanalizujemy definicję operatora **lub-m** pamiętając, że zgodnie z ogólnym założeniem przyjętym w rozdziale 2.8, argumenty wyrażenia **p lub-m q** wyliczamy od lewej do prawej.

- Jeżeli $p = pp$, to rezygnujemy z wyliczenia wartości q (ewaluacja leniwa) i przyjmujemy, że wartość całości jest pp; zauważmy że w tym przypadku być może unikniemy

²⁴ Antycypuję tu przyszłą składnię języka programowania **Lingua**, który będzie mi służył do ilustracji zagadnień związanych z denotacyjnymi modelami języków programowania. Używam w nim angielskich słów kluczowych, by być zgodnym z międzynarodowymi standardami przyjętymi w rzeczywistych językach programowania, a także czcionki Courier New, dla odróżnienia poziomu języka programowania od poziomu metajęzyka jego opisu. W szczególności więc **and** jest napisem z poziomu składni języka, a **oraz** odpowiadającym mu spójnikiem logicznym. Więcej o rozróżnieniu poziomów językowych w rozdz.6.1.1.

pojawienia się sygnału błędu, co właśnie oznacza, że McCarthy'owska alternatywa nie jest transparentna względem błędów.

- Jeżeli $p = ff$, to dla wyliczenia wartości wyrażenia musimy wyliczyć q którego wartość staje się wartością wyrażenia.
- Jeżeli $p = bb$, to oznacza, że wykonywanie programu uległo przerwaniu przy wyliczaniu pierwszego argumentu, a więc do wyliczenia drugiego argumentu już nie doszło. W konsekwencji bb musi być wartością wyrażenia.

Reguła wyliczania wartości koniunkcji jest analogiczna.

Oczywiście dla argumentów klasycznych, wartości spójników McCarthy'ego są zgodne z klasycznymi (obszary zaznaczone na szaro). Implikację McCarthy'ego definiuje się klasyczną równością:

p **implikuje-m** $q = (\text{nie-m } p) \text{ lub-m } q$

Jak się okazuje, nie wszystkie klasyczne tautologie logiczne są w rachunku McCarthy'ego spełnione. Do najważniejszych spełnionych należą:

- **lub-m** i **oraz-m** są łączne,
- prawa de Morgana,

a najważniejsze niespełnione są następujące:

- **lub-m** i **oraz-m** nie są przemienne, np. $ff \text{ oraz-m } bb = ff$, ale $bb \text{ oraz-m } ff = bb$
- **oraz-m** jest rozdzielne względem **lub-m** tylko prawostronnie, tzn.
 $p \text{ oraz-m } (q \text{ lub-m } s) = (p \text{ oraz-m } q) \text{ lub-m } (p \text{ oraz-m } s)$ natomiast
 $(q \text{ lub-m } s) \text{ oraz-m } p \neq (q \text{ oraz-m } p) \text{ lub-m } (s \text{ oraz-m } p)$ gdyż
 $(pp \text{ lub-m } bb) \text{ oraz-m } ff = ff$ i $(pp \text{ oraz-m } ff) \text{ lub-m } (bb \text{ oraz-m } ff) = bb$
- analogicznie **lub-m** jest rozdzielne względem **oraz-m** tylko prawostronnie,
- prawo wyłączonego środka nie jest spełnione, ale $p \text{ lub-m } (\text{nie-m } p)$ nigdy nie jest fałszem,
- prawo sprzeczności nie jest spełnione, ale $p \text{ oraz-m } (\text{nie-m } p)$ nigdy nie jest prawdą.

Na gruncie tak określonego rachunku zdań zostanie w przyszłości (rozdział 8) zbudowany znacznie bogatszy rachunek trójwartościowych predykatów częściowych²⁵, który znajdzie zastosowanie w regułach konstruowania programów poprawnych. Na poziomie rachunku zdań, częściowość predykatów odpowiada sytuacji, gdy bb reprezentuje błąd lub nieskończone obliczenie.

Zauważmy, że tak rozumiany rachunek McCarthy'ego jest, dzięki leniwej ewaluacji, implementowalny, co wynika z równości:

$bb \text{ oraz } p = bb$ dla każdego $p : (pp, ff, bb)$

$bb \text{ lub } p = bb$ dla każdego $p : (pp, ff, bb)$

²⁵ Częściowość predykatów będzie konsekwencją taktu, że w wyrażeniach boolowskich będą mogły występować wywołania procedur.

Jeżeli program zapętli się przy wyliczaniu pierwszego argumentu, to nigdy nie dojdzie do wyliczenia drugiego. Tej własności nie ma alternatywny dla rachunku McCarthy’ego rachunek zdefiniowany przez S.C. Kleene’ego [49], gdzie

ff oraz bb = ff

bb oraz ff = ff

a więc dla sfalsyfikowania koniunkcji wystarczy, by którykolwiek z jej argumentów był fałszywy. Operacyjnie oznacza to, że przy wyliczaniu wartości wyrażenia **p oraz q** trzeba wyliczyć wartości obu argumentów. Taka zasada da się implementacyjnie zrealizować jedynie wtedy, gdy **bb** nie odpowiada obliczeniu nieskończonemu, chyba że moglibyśmy wyliczać wartości obu argumentów równolegle. Jak jednak zobaczymy (rozdział 12), rachunek Kleene’ego znajduje zastosowanie w SQL-owych zapytaniach, a także (rozdział 8.2) częściowo w programowaniu walidującym.

2.10 Algebry danych

Występujące w językach programowania typy danych opisujemy przez określenie zbiorów obiektów, takich jak liczby, wartości boolowskie, tablice, rekordy, listy itp. oraz operacji na tych obiektach. Na przykład, strukturę typu danych liczbowych moglibyśmy opisać za pomocą krotki

AlgLic = (Liczba, twórz.1, plus, minus, razy, dziel) (2.10-1)

która nazwiemy *algebrą liczb*, gdzie zbiór *Liczba*, zwany *nośnikiem algebry*, jest zbiorem wszystkich liczb, a pozostałe elementy to operacje, a więc funkcje, wykonywalne na liczbach i mające wartość liczbowe. Te funkcje nazywamy *konstruktorami algebry*, gdyż służą one do konstruowania jej elementów. Poniższe formuły określają czym są argumenty i wyniki konstruktorów:

twórz-lc.1 :	$\mapsto$ Liczba	
plus	: Liczba x Liczba	$\mapsto$ Liczba
minus	: Liczba x Liczba	$\mapsto$ Liczba
razy	: Liczba x Liczba	$\mapsto$ Liczba
dziel	: Liczba x Liczba	$\rightarrow$ Liczba

(2.10-2)

Funkcja zeroargumentowa **twórz-lc.1** stanowi w tej algebrze *stałą algebry*. Jest to funkcja, która nie ma argumentów, a jej jedyną wartością jest liczba 1, co zapisujemy w następujący sposób:

twórz-lc.1.() = 1

Gdyby nasza algebra miała stanowić model dla języka programowania, to obecność stałej **twórz-lc.1** oznaczałaby, że liczbę jeden możemy wyrazić na poziomie składni w sposób „bezpośredni” pisząc **twórz-lc.1.()**. Liczby dwa w ten sposób wyrazić nie można. Trzeba ją zbudować z jedynek, a więc napisać na przykład:

plus.(twórz-lc.1.(), twórz-lc.1.())

Liczbę 2 tworzymy więc z dwóch jedynek, a liczbę 1 — „z niczego”. Zarówno

twórz-lc.1.()

jak i

```
plus.(twórz-lc.1.(), twórz-lc.1.())
```

są przykładami wyrażeń w tzw. *składni abstrakcyjnej* (rozdział 2.12) naszej algebry. Ponieważ taka składnia byłaby niewygodna dla użytkownika języka, zwykle upraszcza się ją do tzw. *składni konkretnej* (rozdział 4.5), w której napisalibyśmy odpowiednio 1 oraz 1+1.

Zauważmy, że w naszej algebrze dzielenie jest funkcją częściową ze względu na niewykonalność dzielenia przez zero.

Nasza algebra liczb stanowi przykład *algebry abstrakcyjnej*, a lista formuł (2.10-2) jest jej *sygnaturą* (formalne definicje tych pojęć w Rozdz.2.11). Słowo *abstrakcyjna* służy podkreśleniu, że AlgLic to nie dział matematyki zajmujący się rozwiązywaniem równań wielomianowych, ale pewien abstrakcyjny obiekt matematyczny.

Oczywiście w językach programowania, w których występuje typ liczbowy, możemy pisać w wyrażeniach jedynie liczby o skończonych i ograniczonych rozwinięciach dziesiętnych należących do pewnego zbioru *liczb reprezentowalnych* w arytmetyce mikroprocesora²⁶. Jeżeli przez LiczbaR oznaczmy taki właśnie zbiór liczb to sygnatura naszej algebry byłaby następująca:

twórz-lc.lic :	$\mapsto$ LiczbaR	dla lic : LiczbaR
plus	: LiczbaR x LiczbaR $\rightarrow$ LiczbaR	
minus	: LiczbaR x LiczbaR $\rightarrow$ LiczbaR	
minus	: LiczbaR x LiczbaR $\rightarrow$ LiczbaR	
dziel	: LiczbaR x LiczbaR $\rightarrow$ LiczbaR	

W tej algebrze mamy do czynienia z rodziną konstruktorów zeroargumentowych twórz-lc.lic indeksowaną elementami zbioru LiczbaR. Funkcja twórz-lc nie należy do algebry. Jest meta-funkcją służącą do generowania zeroargumentowych konstruktorów algebry.

Zauważmy, że w tej algebrze wszystkie konstruktory niezeroargumentowe są funkcjami częściowymi, gdyż każdy z nich może „wyprowadzać” poza zbiór liczb reprezentowalnych w arytmetyce komputera. Jednakże dopuszczanie w algebrach danych konstruktorów częściowych miałoby dwie wady:

- wada matematyczna — w teorii algebr abstrakcyjnych (o której niżej) zakładamy, że wszystkie operacje algebr są funkcjami całkowitymi; wprowadzenie funkcji częściowych jest możliwe, ale prowadziłoby do bardziej złożonego modelu,
- wada informatyczna — do języka programowania chcemy wbudować mechanizm informowania użytkownika o sytuacjach, gdy jakaś operacja nie może być wykonana.

Dla uniknięcia obu tych wad wprowadzamy do nośników algebry danych błędy abstrakcyjne opisane w rozdziale 2.8. W naszym przypadku w miejsce nośnika LiczbaR wstawiamy nośnik

LiczbaB = LiczbaR | Błąd

gdzie Błąd zawiera wszystkie komunikaty błędów generowane przez nasze operacje. Sygnatura algebry wzbogaconej o błędy abstrakcyjne ma wtedy następującą postać:

²⁶ Zauważmy, że w podręcznikach języków programowania zwykle podaje się jedynie zakres wielkości np. od -2^{63} do $2^{63} - 1$ (por. [39]) nie wspominając już, że liczby o nieskończonych rozwinięciach binarnych, np. $1/3$, lub o rozwinięciach za długich, będą odpowiednio „przycinane”.

twórz-lc.lic:	$\mapsto$ LiczbaB	dla lic : LiczbaR
plus	: LiczbaB x LiczbaB $\mapsto$ LiczbaB	
minus	: LiczbaB x LiczbaB $\mapsto$ LiczbaB	
razy	: LiczbaB x LiczbaB $\mapsto$ LiczbaB	
dziel	: LiczbaB x LiczbaB $\mapsto$ LiczbaB	

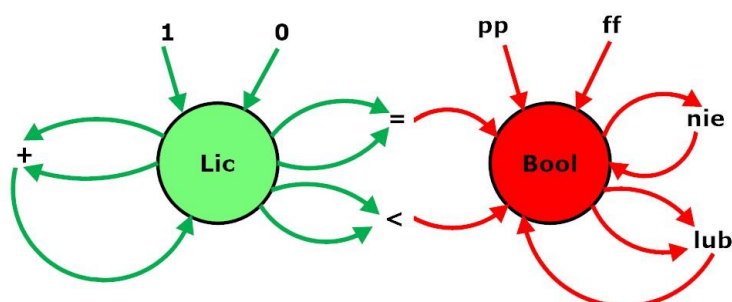
Przechodząc do kolejnego aspektu definiowania struktur danych zauważmy, że w większości języków programowania, z typem danych *liczby* kojarzymy nie tylko operacje arytmetyczne, ale również *predykaty* takie jak np. mniejsze i równe, a więc funkcje o argumentach liczbowych, ale wartościach boolowskich. Aby opisać taką strukturę danych trzeba zbudować algebrę o dwóch nośnikach — LiczbaB i BoolowskaB — odpowiadających dwóm typom danych:

AlgLicBoo = (LiczbaB, BoolowskaB,
 {twórz-lc.lic | lic : LiczbaR}, plus, minus, razy, dziel,
 mniejsze, równe, twórz-bo.pp, twórz-bo.ff, lub, oraz, nie)

Występujące w tej algebrze operacje tworzą następującą sygnaturę:

twórz-lc.lic	:	$\mapsto$ LiczbaB dla lic : LiczbaR	
plus	: LiczbaB x LiczbaB	$\mapsto$ LiczbaB	
minus	: LiczbaB x LiczbaB	$\mapsto$ LiczbaB	
razy	: LiczbaB x LiczbaB	$\mapsto$ LiczbaB	
dziel	: LiczbaB x LiczbaB	$\mapsto$ LiczbaB	
mniejsze	: LiczbaB x LiczbaB	$\mapsto$ BoolowskaB	(2.10-3)
równe	: LiczbaB x LiczbaB	$\mapsto$ BoolowskaB	
twórz-bo.pp	:	$\mapsto$ BoolowskaB	
twórz-bo.ff	:	$\mapsto$ BoolowskaB	
lub	: BoolowskaB x BoolowskaB	$\mapsto$ BoolowskaB	
oraz	: BoolowskaB x BoolowskaB	$\mapsto$ BoolowskaB	
nie	: BoolowskaB	$\mapsto$ BoolowskaB	

Algebrę o dwóch nośnikach nazwiemy *algebrą dwurodzajową*. Czasami sygnaturę algebry opisujemy graficznie jak na Rys. 2.10-1. Dla uproszczenia umieściłem w nim tylko niektóre operacje naszej algebry i przyjąłem jedynie dwie stałe liczbowe.



Rys. 2.10-1 Graficzna reprezentacja sygnatury algebry dwurodzajowej

Zauważmy, że stałe naszej algebry nie wyprowadzają poza zbiór **LiczbaR**. Pozostałe operacje definiujemy również z zachowaniem tej zasady. Na przykład, operacja dodawania byłaby zdefiniowana w sposób następujący:

plus.(lic-1, lic-2) =

lic-1 : Błąd → lic-1

lic-2 : Błąd → lic-2

nie +.(lic-1, lic-2) : LiczbaR → 'przekroczenie-zakresu'²⁷

prawda → +.(lic-1, lic-2)

gdzie „+” oznacza arytmetyczną operację dodawania. Dzięki tej zasadzie, wszystkie elementy naszej algebry, które dają się skonstruować za pomocą jej konstruktorów, należą do zbioru **LiczbaR** | **Błąd**. Zauważmy, że tak zdefiniowane dodawanie nie jest przemienne, gdyż

plus.(błd-1, błd-2) ≠ plus.(błd-2, błd-1)

jeżeli tylko błd-1 ≠ błd-2.

2.11 Algebry wielorodzajowe

O algebrze **AlgLicBoo** mówimy, że jest *algebrą dwurodzajową* o dwóch rodzajach nośników, a mianowicie **LiczbaB** i **BoolowskaB**. Ogólnie będziemy posługiwać się pojęciem *algebry wielorodzajowej* (ang. *many-sorted algebra*) — którą dalej będziemy nazywać po prostu *algebrą* — i którą definiujemy jako krotkę:

$\text{Alg} = (\text{Syg}, \text{Noś}, \text{Kon}, \text{noś}, \text{kon})$

gdzie:

- $\text{Syg} = (\text{Nn}, \text{Nk}, \text{ar}, \text{ro})$ — jest nazywana *sygnaturą algebry*,
- Nn — jest skończonym zbiorem napisów będących nazwami nośników; nośniki algebry nazywamy też jej *rodzajami*,
- Nk — jest skończonym zbiorem nazw konstruktorów algebry,
- $\text{ar} : \text{Nk} \mapsto \text{Nn}^*$ — każdej nazwie konstruktora nk jest przypisany skończony, być może pusty i być może z powtórzeniami ciąg nazw rodzajów $\text{ar.nk} = (\text{nn}_1, \dots, \text{nn}_n)$ zwany *arnością*²⁸ tego konstruktora,
- $\text{ro} : \text{Nk} \mapsto \text{Nn}$ — każdej nazwie konstruktora nk jest przypisana nazwa nośnika ro.nk zwana *rodzajem* tego konstruktora,
- Noś — jest rodziną zbiorów zwanych *nośnikami algebry*,
- Kon — jest rodziną konstruktorów (funkcji całkowitych) o argumentach i wartościach należących do nośników,

²⁷ Występujący w tej klauzuli operator negacji **nie** piszę pismem pogrubionym, gdyż jest to operator z poziomu metajęzyka, a nie konstruktor boolowski algebry **AlgLicBoo**.

²⁸ Termin „arność funkcji” pochodzi od takich określeń jak „unarna” czyli jednoargumentowa, „binarna”, czyli dwuargumentowa, „n-arna” czyli o n argumentach. Arność określa jednak nie tylko liczbę argumentów, ale również ich rodzaje.

$\text{noś} : \text{Nn} \mapsto \text{Noś}$	—	jest funkcją, która każdej nazwie nośnika nn przypisuje nośnik $\text{noś}.nn$,
$\text{kon} : \text{Nk} \mapsto \text{Kon}$	—	jest funkcją, która każdej nazwie konstruktora nk takiej, że $\text{ar}.nk = (nn_1, \dots, nn_n)$ oraz $\text{ro}.nk = nn$ przypisuje funkcję całkowitą $\text{kon}.nk : \text{noś}.nn_1 \times \dots \times \text{noś}.nn_n \mapsto \text{noś}.nn$

Pojęcia „arności” i „rodzaju” odnosimy nie tylko do symboli będących nazwami konstruktorów, ale też i do samych konstruktorów, których te symbole są nazwami. Konstruktory zeroargumentowe, to jest konstruktory o arności będącej ciągiem pustym, nazywamy *stałymi algebrami*. Jeżeli f jest takim konstruktorem to piszemy:

$$f : \mapsto \text{Nośnik}$$

a jedyną wartość tego konstruktora zapisujemy jako

$$f.()$$

Podkreślenia wymaga fakt, że wszystkie konstruktory algebry wielorodzajowej muszą być funkcjami całkowitymi. Jest to założenie techniczne, które w naszych zastosowaniach da się zawsze zrealizować przez wprowadzenie błędów abstrakcyjnych, o których była mowa w rozdziale 2.8.

Ta przydługa definicja została w dużej mierze wprowadzona po to, aby w algebrach wielorodzajowych — które staną się dla nas modelami nie tylko struktur danych, ale też języków programowania i aplikacji informatycznych — w sposób wyraźny odróżnić warstwę syntaktyczną (składniową), którą jest sygnatura, od warstwy znaczeniowej (denotacyjnej), do której należą dziedziny i określone na nich funkcje. Dla konkretnych algebr, takich jak opisana w rozdziale 2.10 algebra AlgLicBoo , sygnatura jest opisywana zwykle układem formuł takich jak (2.10-3). Rozważmy teraz dwie algebry

$$\text{Alg}_i = (\text{Syg}_i, \text{Noś}_i, \text{Kon}_i, \text{noś}_i, \text{kon}_i) \text{ dla } i = 1, 2$$

z odpowiednimi sygnaturami

$$\text{Syg}_i = (\text{Nn}_i, \text{Nk}_i, \text{ar}_i, \text{ro}_i) \text{ dla } i = 1, 2$$

Powiemy, że sygnatura Syg_2 jest *rozszerzeniem sygnatury* Syg_1 lub Syg_1 jest *ograniczeniem sygnatury* Syg_2 jeżeli

1. $\text{Nn}_1 \subset \text{Nn}_2$ oraz $\text{Nk}_1 \subset \text{Nk}_2$,
2. funkcje ar_2, ro_2 pokrywają się z funkcjami ar_1, ro_1 na Nk_1 .

Powiemy, że algebra Alg_2 jest *rozszerzeniem algebry* Alg_1 lub Alg_1 jest *ograniczeniem algebry* Alg_2 jeżeli

1. Syg_2 jest rozszerzeniem sygnatury Syg_1 ,
2. dla każdego rodzaju $nn : \text{Nn}_1$ zachodzi $\text{noś}_1.nn \subset \text{noś}_2.nn$,
3. dla każdego symbolu konstruktora $nk : \text{Nk}_1$ konstruktor $\text{kon}_2.nk$ pokrywa się z konstruktorem $\text{kon}_1.nk$ na odpowiednich nośnikach algebry Alg_1 .

Innymi słowy (nietrywialne) rozszerzenie algebry powstaje przed dodanie do niej nowych nośników i/lub nowych funkcji i/lub nowych elementów do istniejących nośników.

Dwie algebry wielorodzajowe nazwiemy *podobnymi*, jeżeli mają tę samą sygnaturę. W przyszłości będziemy często konstruowali konkretne algebry wielorodzajowe definiując ich nośniki i konstruktory, ale nie definiując *explicite* sygnatur. W takich przypadkach powiemy, że algebry są *podobne*, gdy daje się dla nich zbudować wspólną sygnaturę.

Jeżeli Alg_1 i Alg_2 są podobne, to powiemy, że Alg_1 jest *podalgebrą* algebry Alg_2 lub Alg_2 jest *nadalgebrą* algebry Alg_1 gdy

1. nośniki algebry Alg_1 są podzbiorami odpowiadających im nośników algebry Alg_2 ,
2. operacje algebry Alg_2 pokrywają się z odpowiadających im operacjami algebry Alg_1 , na nośnikach algebry Alg_2 .

Zatem każda nadalgebra jest rozszerzeniem, ale nie na odwrót. *Homomorfizmem wielorodzajowym* z algebry Alg_1 w podobną do niej algebrę Alg_2 gdzie

$$Alg_i = (S_{y_i}, Noś_i, Kon_i, noś_i, kon_i), i = 1, 2$$

nazwiemy rodzinę funkcji

$$H = \{h.nn \mid nn : N_n\}$$

której elementy nazywamy *składowymi homomorfizmu*, taką że:

$$h.nn : noś_1.nn \mapsto noś_2.nn \text{ dla wszystkich } nn : N_n,$$

a więc gdzie każda funkcja z tej rodziny elementy nośnika Alg_1 odwzorowuje w elementy nośnika Alg_2 w taki sposób, że dla każdego symbolu konstruktora $nk : N_k$ takiego, że

$$ar.nk = (nn_1, \dots, nn_n) \text{ gdzie } n \geq 0$$

i dla każdej krotki argumentów

$$(a_1, \dots, a_n) : noś_1.nn_1 \times \dots \times noś_1.nn_n$$

zachodzi równość

$$h.nn.(kon_1.nk.(a_1, \dots, a_n)) = kon_2.nk.(h.nn_1.a_1, \dots, h.nn_n.a_n) \quad (2.11-1)$$

Tak więc homomorficznym obrazem

wartości funkcji $kon_1.nk$ pierwszej algebry dla krotki (być może pustej) argumentów $(a_1, \dots, a_n)$

jest wartość odpowiadającej jej funkcji $kon_2.nk$ drugiej algebry dla homomorficznych obrazów tych argumentów, a więc krotki $(h.nn_1.a_1, \dots, h.nn_n.a_n)$.

Zauważmy, że dla $n=0$ równość (2.11-1) przybiera postać:

$$h.nn.(kon_1.nk.()) = kon_2.nk.()$$

Fakt, że H jest homomorfizmem z Alg_1 w Alg_2 oznaczamy symbolicznie pisząc

$$H : Alg_1 \mapsto Alg_2$$

Z naszej definicji homomorfizmu wynika, że jeżeli jakieś nośniki algebry Alg_1 są puste, to odpowiadające im składowe homomorfizmu muszą być funkcjami pustymi. Algebrę, której wszystkie nośniki są puste nazwiemy *algebrą pustą*.

W ogólnym przypadku homomorfizm nie odwzorowuje Alg_1 „na” Alg_2 , ale „w” Alg_2 , co oznacza, że nie każdy element Alg_2 musi mieć swój przeciwobraz w Alg_1 . Tę własność ma na przykład identycznościowy homomorfizm

$$\text{Id} : (\text{Całkowita}, 1, \text{plus}, \text{minus}) \mapsto (\text{Liczba}, 1, \text{plus}, \text{minus})$$

z jednorodzącej algebry liczb całkowitych w jednorodzącej algebrę liczb. Przykładem homomorfizmu „na” jest nieidentycznościowy homomorfizm odwzorowujący liczby całkowite w parzyste:

$$\text{Pa} : (\text{Całkowita}, 1, \text{plus}, \text{minus}) \mapsto (\text{Parzysta}, 2, \text{plus}, \text{minus})$$

i określony równością $\text{Pa}.c = 2c$. W ogólnym przypadku homomorfizm $H : \text{Alg}_1 \mapsto \text{Alg}_2$ nazwiemy:

- *monomorfizmem* albo *włożeniem* — jeżeli wszystkie jego składowe są funkcjami różnowartościowymi; przykładami są homomorfizmy Id oraz Pa ,
- *epimorfizmem* albo *nałożeniem* — jeżeli wszystkie jego składowe są funkcjami „na”; przykładem jest homomorfizm Pa ,
- *izomorfizmem* — jeżeli jest jednocześnie monomorfizmem i epimorfizmem; przykładem jest homomorfizm Pa .

Twierdzenie 2.11-1 Dla każdego homomorfizmu $H : \text{Alg}_1 \mapsto \text{Alg}_2$ obraz Alg_1 w Alg_2 , czyli ograniczenie Alg_2 do zbiorów będących obrazami — ze względu na H — dziedzin Alg_1 wraz z odpowiednim ograniczeniem operacji algebry Alg_2 tworzy podalgebrę algebry Alg_2 . ■

Dowód Żeby udowodnić to twierdzenie wystarczy pokazać, że obrazy w Alg_2 dziedzin algebry Alg_1 są zamknięte ze względu na jej operacje. Weźmy więc dowolną krotkę argumentów $(b_1, \dots, b_n)$ z algebry Alg_2 , które mają swoje przeciwobrazy $(a_1, \dots, a_n)$ w algebrze Alg_1 , tzn. dla których zachodzi:

$$(b_1, \dots, b_n) = (h.nn_1.a_1, \dots, h.nn_n.a_n)$$

Niech teraz dla pewnego symbolu operacji nk zachodzi

$$\text{kon}_2.nk.(b_1, \dots, b_n) = b$$

Należy pokazać, że b ma swój przeciwobraz w Alg_1 . Jest tak rzeczywiście, gdyż na mocy (2.11-1)

$$\text{kon}_2.nk.(b_1, \dots, b_n) = \text{kon}_2.nk.(h.nn_1.a_1, \dots, h.nn_n.a_n) = h.nn.(\text{kon}_1.nk.(a_1, \dots, a_n))$$

a więc $(\text{kon}_1.nk.(a_1, \dots, a_n))$ jest szukanym przeciwobrazem b w Alg_1 . ■

Algebrę będącą obrazem homomorfizmu $H : \text{Alg}_1 \mapsto \text{Alg}_2$ nazywamy *jądrem homomorfizmu* H w Alg_2 .

Wszystkie rozumowania dotyczące homomorfizmów można uogólnić na przypadek, gdy sygnatury dwóch algebr

$$\text{Syg}_i = (Nn_i, Nk_i, ar_i, ro_i) \text{ dla } i = 1, 2$$

co prawda nie są jednakowe, ale są *podobne* w tym sensie, że istnieją dla nich dwie różnowartościowe funkcje podobieństwa:

$$Pn : Nn_1 \mapsto Nn_2$$

$$Pk : Nk_1 \mapsto Nk_2$$

takie, że jeżeli

$$Pk.nk_1 = nk_2$$

$$ar_1.nk_1 = nn_{11}, \dots, nn_{1p}$$

$$ar_2.nk_2 = nn_{21}, \dots, nn_{2m}$$

to

$$p = m$$

$$Pr.nn_{1i} = nn_{2i} \quad \text{dla } i = 1;p$$

Mówiąc prościej, dwie sygnatury są podobne, jeżeli mają tę samą liczbę rodzajów i nazw funkcji, a odpowiadające sobie nazwy funkcje mają takie same arności i rodzaje z dokładnością do nazw rodzajów.

Możemy teraz uogólnić pojęcie podobieństwa algebr: dwie algebry nazwiemy *podobnymi*, gdy ich sygnatury są podobne. Przy każdych ustalonych funkcjach Pn i Pk pojęcie homomorfizmu i związane z nim twierdzenia przenoszą na tak uogólnione pojęcie algebr podobnych.

2.12 Składnia abstrakcyjna i algebry osiągalne

Każda sygnatura

$$Syg = (Nn, Nk, ar, ro)$$

jednoznacznie wyznacza pewną algebrę o tej sygnaturze, której nośnikami są języki formalne. Tę algebrę będziemy nazywali *składnią abstrakcyjną nad sygnaturą* Syg i oznaczali $SkAbs(Syg)$ ²⁹. Elementami jej nośników są wyrażenia (słowa) składające się na język wielorodzajowy

$$\{Wyr.nn \mid nn : Nn\}$$

zdefiniowany gramatyką równaniową (por. rozdział 2.5) w opisany niżej sposób.

Każdej nazwie nośnika nn przypisuję *składniową kategorię wyrażen typu* nn , którą oznaczam przez $Wyr.nn$. Krotkę tych wszystkich języków (kategorii) definiujemy gramatyką równaniową, w której każdej nazwie rodzaju $nn : Nn$ odpowiada jedno równanie³⁰:

$$\begin{aligned} Wyr.nn &= \{nk_1\} \odot \{\{\} \odot Wyr.nn_{11} \odot \{\}, \} \odot \dots \odot \{\{\} \odot Wyr.nn_{1n(1)} \odot \{\}\} \mid \\ &\dots \\ &\{nk_k\} \odot \{\{\} \odot Wyr.nn_{k1} \odot \{\}, \} \odot \dots \odot \{\{\} \odot Wyr.nn_{kn(k)} \odot \{\}\} \end{aligned} \quad (2.12-1)$$

Symbole funkcyjne nk_i dla $i = 1;k$ są wszystkimi takimi symbolami, dla których

$$ro.nk_i = nn$$

oraz

$$ar.nk_i = (nn_{i1}, \dots, nn_{in(i)}) \quad \text{dla } i = 1;k$$

²⁹ Pojęcie składni abstrakcyjnej jako matematycznej idealizacji składni języków programowania pojawiło się po raz pierwszy w pracach J. McCarthy'ego [56] i P. Landina [51], a z algebrami wielorodzajowymi zostało powiązane przez czwórkę autorów J.A. Goguen, J.W. Thatcher, E.G. Wagner i J.B. Wright [44]. Nieco później użyłem tego pojęcia przy tworzeniu formalnej definicji podzbioru języka programowania Pascal [21].

³⁰ Zakładamy oczywiście, że przecinki „,” oraz nawiasy „(” i „)” nie występują w sygnaturze algebry jako nazwy konstruktorów.

Zakładamy, że jeżeli dla jakiegoś rodzaju nn nie istnieje żadna funkcja tego rodzaju, to odpowiadający jej język jest pusty, a więc jest zdefiniowany równaniem

$$\text{Wyr}.nn = \emptyset$$

To założenie jest niezbędne, aby składnia abstrakcyjna zbudowana dla zadanej sygnatury dała się przedstawić jako algebra nad tą właśnie sygnaturą. Słowa każdego niepustego języka $\text{Wyr}.nn$ są postaci

$$nk_i(w_{i1}, \dots, w_{in(i)})$$

tzn. postaci $nk_i \circ (\circ w_{i1} \circ \dots \circ w_{in(i)} \circ)$ gdzie $\circ$ jest konkatenacją słów oraz

$$w_{ik} : \text{Wyr}.nn_k.$$

W tym miejscu warto zauważyć, że jeżeli w sygnaturze nie występują nazwy stałych, a więc nazwy funkcji o arnościach zerowych, to wszystkie nośniki (języki) odpowiadającej jej składni abstrakcyjnej są puste.

Skoro składnie abstrakcyjne przypisujemy sygnaturom, to możemy je też przypisywać algébrom (przez ich sygnatury). Jeżeli algebra Alg ma sygnaturę Syg , to o $\text{SkAbs}(\text{Syg})$ powiemy, że jest *składnią abstrakcyjną algebry* Alg . Na przykład, dziedziny składni abstrakcyjnej dla algebry AlgLicBoo z rozdziału 2.10 są opisane następującą gramatyką równaniową gdzie WyrLic oraz WyrBoo oznaczają język wyrażeń liczbowych i odpowiednio boolowskich:

$$\begin{aligned} \text{WyrLic} = & 0 \mid 1 \mid \\ & \text{plus}(\text{WyrLic}, \text{WyrLic}) \mid \text{minus}(\text{WyrLic}, \text{WyrLic}) \mid \\ & \text{razy}(\text{WyrLic}, \text{WyrLic}) \mid \text{dziel}(\text{WyrLic}, \text{WyrLic}) \end{aligned} \quad (2.12-1)$$

$$\begin{aligned} \text{WyrBoo} = & pp \mid ff \mid \\ & \text{mniejsze}(\text{WyrLic}, \text{WyrLic}) \mid \text{równe}(\text{WyrLic}, \text{WyrLic}) \mid \\ & \text{lub}(\text{WyrBoo}, \text{WyrBoo}) \mid \text{oraz}(\text{WyrBoo}, \text{WyrBoo}) \mid \text{nie}(\text{WyrBoo}) \end{aligned}$$

W tej gramatyce równaniowej zastosowałem trzy konwencje notacyjne, które przyjmuję jako standard na przyszłość:

1. ilekroć nie prowadzi to do nieporozumień, to w miejsce zbioru jednoargumentowego $\{a\}$ piszę po prostu odpowiadający mu element a ,
2. dla każdego konstruktora zeroargumentowego o nazwie nk zamiast $nk()$ piszę nk , np. zamiast $1()$ piszę 1 ,
3. opuszczam znaki konkatenacji „ $\circ$ ”, a więc zamiast $a \circ b$ piszę ab .

A oto przykłady wyrażeń, odpowiednio liczbowego i boolowskiego, z tej algebry

$$\begin{aligned} & \text{plus}(\text{plus}(\text{minus}(1,0),1),\text{plus}(1,1)) \\ & \text{lub}(\text{mniejsze}(\text{plus}(\text{plus}(\text{minus}(1,0),1),\text{plus}(1,1)),\text{plus}(1,1)),ff) \end{aligned}$$

Jak widzimy, wyrażenia naszej algebry nie zawierają zmiennych i są pisane w tzw. *notacji prefiksowej*, gdzie znak operacji poprzedza jej argumenty, a więc gdzie zamiast pisać $(1 \text{ plus } 1)$ jak w *notacji infiksowej*, piszemy $\text{plus}(1,1)$.

Elementami języków (nośników) odpowiadającej tej gramatyce algebry są wyrażenia liczbowe i boolowskie bez zmiennych, a operacjami — konstruktory wyrażeń odpowiadające nazwom konstruktorów z naszej sygnatury. Na przykład, symbolowi „plus” pochodzącemu z

sygnatury algebry AlgLicBoo odpowiada konstruktor algebry SkAbs(Syg), który oznaczę przez [plus] i zdefiniuję równaniem:

$$[\text{plus}].[\text{wyr-lic1}, \text{wyr-lic2}] = \text{plus}(\text{wyr-lic1}, \text{wyr-lic2})^{31}$$

Jest to więc operator, który z dwóch wyrażeń liczbowych wyr-lic1 i wyr-lic2 tworzy nowe wyrażenie:

$$\text{plus}(\text{wyr-lic1}, \text{wyr-lic2}).$$

Możemy teraz sformułować twierdzenie o podstawowym znaczeniu dla denotacyjnych modeli języków programowania.

Twierdzenie 2.12-1 *Dla każdej wielorodzajowej algebry Alg o sygnaturze Syg istnieje dokładnie jeden homomorfizm $H : \text{SkAbs}(\text{Syg}) \mapsto \text{Alg}$.* ■

Dowód Każdy homomorfizm $H : \text{SkAbs}(\text{Sig}) \mapsto \text{Alg}$ musi z definicji spełniać równania

$$H.\text{nn}.[\text{nk}(\text{wyr}_1, \dots, \text{wyr}_n)] = \text{kon}.\text{nk}.[H.\text{nn}_1.\text{wyr}_1, \dots, H.\text{nn}_n.\text{wyr}_n]$$

gdzie

$$\text{ar}.\text{nk} = (\text{nn}_1, \dots, \text{nn}_n)$$

$$\text{ro}.\text{nk} = \text{nn}$$

$$\text{wyr}_i : \text{Wyr}.\text{nn}_i \text{ dla } i = 1, 2, \dots, n$$

Ponieważ każde wyrażenie należące do składni abstrakcyjnej daje się jednoznacznie przedstawić w postaci $\text{nk}(\text{wyr}_1, \dots, \text{wyr}_n)$, więc powyższe równania dobrze i jednoznacznie definiują rodzinę funkcji $\{H.\text{nn} \mid \text{nn}:\text{Nn}\}$. W przypadku pustych nośników algebry składni odpowiadające im składowe homomorfizmy są puste. ■

Ten jednoznacznie wyznaczony homomorfizm H będę nazywał *homomorfizmem konstytuującym* algebry Alg, gdyż w jakimś sensie wyznacza on jednoznacznie tę algebrę. Na przykład, jeżeli przez $\{L, B\}$ oznaczymy homomorfizm konstytuujący dla algebry AlgLicBoo, to ten homomorfizm odwzorowuje wyrażenie boolowskie $\text{mniejsze}(\text{plus}(1,1), \text{razy}(1,1))$ w wartość boolowską ff:

$$\begin{aligned} B.[\text{mniejsze}(\text{plus}(1,1), \text{razy}(1,1))] &= \\ \text{kon}.\text{mniejsze}.(L.[\text{plus}(1,1)], L.[\text{razy}(1,1)]) &= \\ \text{kon}.\text{mniejsze}.(\text{kon}.\text{plus}.(L.[1], L.[1]), \text{kon}.\text{razy}.(L.[1], L.[1])) &= \\ \text{kon}.\text{mniejsze}(\text{kon}.\text{plus}(1,1), \text{kon}.\text{razy}(1,1)) &= \text{ff} \end{aligned}$$

Na mocy Tw.2.11-1 i Tw.2.12-1, w każdej algebrze Alg istnieje podalgebra będąca jądrem jej (jedyne) homomorfizmu konstytuującego. Tę podalgebrę nazywamy *podalgebrą osiągalną* algebry Alg. Nazwa *osiągalna* pochodzi stąd, że każdy element tej algebry daje się skonstruować (osiągnąć) ze stałych algebry przez zastosowanie do nich konstruktorów algebry. Na przykład, podalgebrą osiągalną algebry

$$(\text{Liczba}, 1, \text{plus}, \text{dziel})$$

jest algebra liczb wymiernych dodatnich

$$(\text{WymDod}, 1, \text{plus}, \text{dziel}),$$

³¹ Wprowadzam tu metanawiasy kwadratowe, by odróżnić je od nawiasów zwykłych występujących w definiowanym języku.

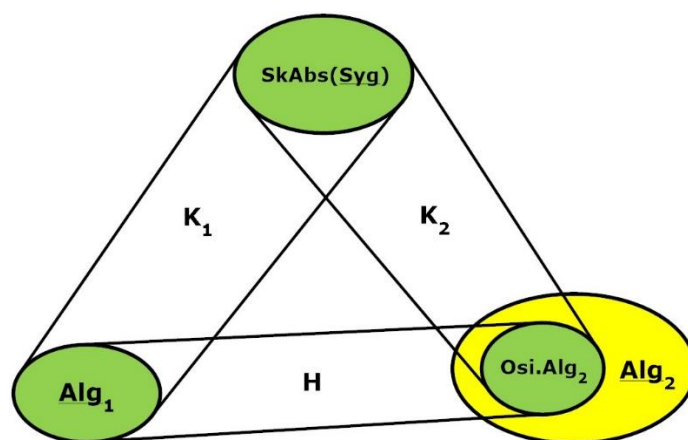
gdyż stosując do liczby 1 operacje arytmetyczne można skonstruować jedynie liczby wymierne dodatnie. Natomiast, gdybyśmy usunęli z niej stałą 1, to nośniki jej części osiągalnej byłyby zbiorami pustymi. Oczywiście składnia abstrakcyjna algebry o pustej części osiągalnej jest również pusta.

O algebrze Alg powiemy, że jest *algebrą osiągalną*, jeżeli jest ona identyczna ze swoją częścią osiągalną. W szczególności każda składnia abstrakcyjna jest algebrą osiągalną. Osiągalną jest również algebra pusta. Teraz możemy sformułować dwa ważne twierdzenia:

Twierdzenie 2.12-2 *Dla każdych dwóch algebr podobnych Alg_1 i Alg_2 , jeżeli Alg_1 jest osiągalna, to istnieje co najwyżej jeden homomorfizm*

$$H : Alg_1 \mapsto Alg_2,$$

a jeżeli tak jest, to obraz Alg_1 w Alg_2 jest osiągalny. ■



Rys. 2.12-1 Algebry osiągalne

Dowód Twierdzenie to, a także jego dowód, ilustruje Rys. 2.12-1. Skoro występujące w twierdzeniu algebry są podobne, to znaczy, że mają wspólną sygnaturę Syg oraz wspólną składnię abstrakcyjną SkAbs(Syg). Stąd, na mocy Tw.2.12.1, istnieją dwa jednoznacznie wyznaczone homomorfizmy konstytuujące

$$K_1 : \text{SkAbs}(\text{Syg}) \mapsto Alg_1 \text{ oraz}$$

$$K_2 : \text{SkAbs}(\text{Syg}) \mapsto Alg_2$$

Jeżeli teraz założymy, że istnieje homomorfizm $H : Alg_1 \mapsto Alg_2$, to złożenie

$$K_1 \bullet H : \text{SkAbs}(\text{Syg}) \mapsto Alg_2.$$

określone jako złożenie odpowiednich składowych tych homomorfizmów, jest homomorfizmem. Ponieważ jednak K_2 jest jedynym homomorfizmem z $\text{SkAbs}(\text{Syg})$ do Alg_2 , musi zachodzić równość

$$K_1 \bullet H = K_2,$$

a skoro Alg_1 jest osiągalna, to powyższe równanie wyznacza H w sposób jednoznaczny, gdyż, gdyby istniał inny homomorfizm spełniający to równanie, to można by zbudować inny homomorfizm z $\text{SkAbs}(\text{Syg})$ w Alg_2 , co jednak na mocy Tw. 2.12-1 nie jest możliwe. Z tego równania wynika też, że obraz Alg_1 w Alg_2 jest osiągalny. ■

Jako natychmiastową konsekwencję tego twierdzenia otrzymujemy kolejne twierdzenie:

Twierdzenie 2.12-3 Dla dowolnej niepustej algebry Alg nad sygnaturą Syg są równoważne następujące tezy:

- (1) Alg jest osiągalna,
- (2) każdy homomorfizm typu $H : Alg_1 \mapsto Alg$ (dla dowolnej Alg_1) jest nałożeniem,
- (3) konstytuujący homomorfizm $K : SkAbs(Syg) \mapsto Alg$ jest nałożeniem. ■

Dowód Załóżmy, że Alg jest osiągalna. Niech dla pewnej Alg_1 podobnej do Alg istnieje homomorfizm

$$H : Alg_1 \mapsto Alg,$$

i niech

$$K : SkAbs(Syg) \mapsto Alg_1$$

będzie homomorfizmem konstytuującym dla Alg_1 . Wtedy

$$K \bullet H : SkAbs(Syg) \mapsto Alg$$

jest homomorfizmem konstytuującym dla Alg, więc skoro Alg jest osiągalna, to $K \bullet H$ jest nałożeniem, a więc i H musi być nałożeniem. Tak więc (1) implikuje (2). (3) wynika z (2) jako jej przypadek szczególny, natomiast (2) implikuje (1) na mocy definicji osiągalności algebry. ■

Na koniec jeszcze jedno użyteczne pojęcie i związane z nim twierdzenie.

Twierdzenie 2.12-4 Algebra ma niepustą część osiągalną, wtedy i tylko wtedy, gdy istnieje w niej co najmniej jedna stała czyli konstruktor zeroargumentowy. ■

Dowód Jeżeli w algebrze istnieje konstruktor zeroargumentowy, to jego wartość należy do części osiągalnej algebry, a więc ta część jest niepusta. Jeżeli natomiast konstruktor zeroargumentowy nie istnieje, to w gramatyce równaniowej odpowiadającej naszej algebrze nie występują jednomiany stałe, co oznacza, że gramatyka ta generuje wektor języków pustych, a więc wszystkie nośniki algebry składni abstrakcyjnej są puste. Stąd wprost z definicji części osiągalnej wynika, że jest ona pusta. ■

Składnie abstrakcyjne są z reguły dość nieporęczne w użyciu i dlatego — jak zobaczymy w rozdziale 4.5 — zastępuje się je homomorficznymi do nich składniami, które historycznie nazywano *konkretnymi*. W takim przypadku słowa będące elementami składni abstrakcyjnej odpowiadają *drzewom parsingu* składni konkretnej, które są podstawowym pojęciem teorii kompilacji języków programowania (por. np. [3]).

2.13 Algebry jednoznaczne i wieloznaczne

Algebrę Alg nad sygnaturą Syg nazwiemy *jednoznaczną* jeżeli jej homomorfizm konstytuujący

$$K : SkAbs(Syg) \mapsto Alg$$

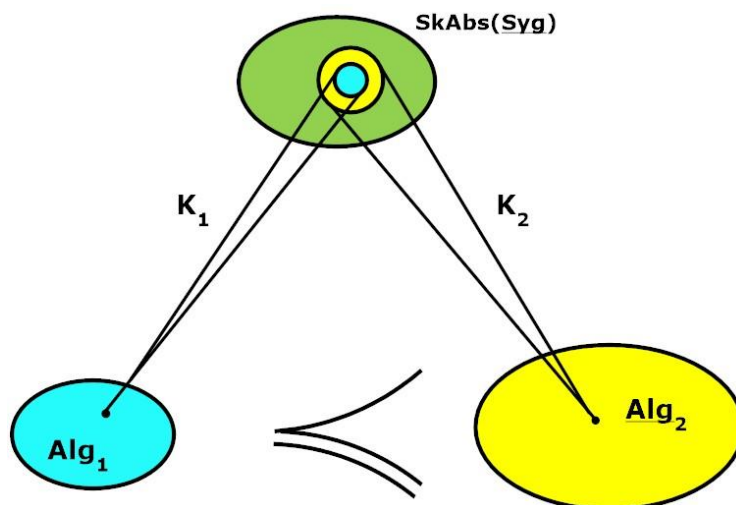
jest różnowartościowy, tj. jeżeli dla każdej nazwy dziedziny nn i każdego elementu e z Alg należącego do dziedziny Noś.nn istnieje co najwyżej jedno wyrażenie $w : Wyr.nn$ w algebrze $SkAbs(Syg)$ takie, że

$$K.nn.w = e$$

Algebrę nazwiemy *wieloznaczną*, jeżeli nie jest jednoznaczna.

Algebry denotacji języków programowania najczęściej nie są jednoznaczne. Na przykład, opisana w Rozdz.2.10 arytmetyka rzeczywista AlgLic (gdyby uzupełnić ją o błędy

abstrakcyjne, aby uczynić jej konstruktory całkowitymi) nie jest jednoznaczna, bo np. dwa różne wyrażenia $\text{plus}(\text{plus}(1,1),1)$ oraz $\text{plus}(1,\text{plus}(1,1))$ odpowiadają tej samej liczbie 3 w AlgLic.



Rys. 2.13-1 Dwie algebry wieloznaczne

Powiemy, że algebra Alg_1 jest *mniej lub tak samo wieloznaczna* niż algebra Alg_2 , co zapiszemy jako

$$\text{Alg}_1 \preceq \text{Alg}_2$$

gdy homomorfizm K_2 nie skleja więcej niż K_1 (Rys. 2.13-1 Dwie algebry wieloznaczne) to znaczy, gdy dla każdego dwóch wyrażen abstrakcyjnych w_1 i w_2 tego samego rodzaju nn zachodzi implikacja

$$\text{jeżeli } K_1.nn.w_1 = K_1.nn.w_2 \text{ to } K_2.nn.w_1 = K_2.nn.w_2 \quad (2.13-1)$$

Ten warunek można intuicyjnie wyrazić w ten sposób, że jeżeli jakiś element algebry Alg_1 daje się zbudować na dwa różne sposoby — tymi „sposobami” są dwa wyrażenia abstrakcyjne w_1 i w_2 — to te dwa „sposoby” tworzą ten sam element algebry Alg_2 .

Algebry wieloznaczne odgrywają ważną rolę w teorii języków programowania, gdyż algebry składni konkretnej istniejących języków programowania — gdyby je formalnie opisano — najczęściej okazałyby się wieloznaczne. Dla wyjaśnienia tego zjawiska założmy, że $\text{SkAbs}(\text{Syg})$ jest składnią abstrakcyjną określoną równaniem

$$\text{WyrLic} = 0 \mid 1 \mid +(\text{WyrLic}, \text{WyrLic})$$

Alg_1 jest algebrą infiksowych wyrażen bez nawiasów określoną gramatyką

$$\text{WyrLic} = 0 \mid 1 \mid \text{WyrLic} + \text{WyrLic}$$

natomiast Alg_2 jest algebrą liczb całkowitych. Niech teraz K_1 zamienia prefiksy na infiksy i usuwa nawiasy, natomiast K_2 odwzorowuje napisy w liczby.

Antycypując rozważania rozdziału 4 algebrę liczb nazwiemy *algebrą denotacji* (znaczeń) dla obu naszych algebr wyrażen liczbowych, a homomorfizm K_2 — *homomorfizmem denotacyjnym* (semantyką) algebry składni abstrakcyjnej. Powstaje pytanie, czy dla algebry beznawiasowej Alg_2 można również zbudować homomorfizm denotacyjny w algebrę liczb?

Aby na to pytanie odpowiedzieć zauważmy, że dla tak określonych algebr i odpowiadających im homomorfizmów mamy następujące równości:

$$K_1.[+(+(1,1),1)] = 1+1+1$$

$$K_2.[+(+(1,1),1)] = 3$$

$$K_1.[+(1,+(1,1))] = 1+1+1$$

$$K_2.[+(1,+(1,1))] = 3$$

Jak więc widzimy K_1 skleja nie więcej niż K_2 . W matematycznej praktyce, a więc i w językach programowania, często opuszczamy „zbędne nawiasy” wykorzystując łączność „nawiasowanych” operacji³². Algebry odpowiadające tym językom nie są jednoznaczne, a więc nie są izomorficzne ze składnią abstrakcyjną, co w przypadku ogólnym nie gwarantuje im istnienia homomorfizmu denotacyjnego. Jeżeli jednak nie są bardziej wieloznaczne od algebry denotacji, to taki homomorfizm istnieje, co wynika z następującego twierdzenia:

Twierdzenie 2.13-1 *Jeżeli Alg_1 i Alg_2 są podobne i Alg_1 jest osiągalna, to (jedyne) homomorfizm $D : Alg_1 \mapsto Alg_2$ istnieje wtedy i tylko wtedy, gdy $Alg_1 \preceq Alg_2$. ■*

Ten jedyny homomorfizm można skonstruować jako (wyrażając to niezbyt precyzyjnie) złożenie odwrotności K_1 z K_2 , czyli

$$D = K_1^{-1} \bullet K_2.$$

Co prawda odwrotność K_1 odwzorowuje elementy Alg_1 w zbiory wyrażeń, jednakże wszystkie te wyrażenia homomorfizm K_2 odwzoruje w ten sam element algebry Alg_2 . To oczywiście nie dowód, ale — jak mawiają matematycy — jedynie „agitacja za dowodem”. Czytelnika zainteresowanego dowodem odsyłam do [24].

Przydatność wieloznacznych gramatyk języków programowania, również z punktu widzenia parsingu, była rozważana już w roku 1972 przez A.V. Aho i J.D. Ullmana w pracy [3].

2.14 Algebry i gramatyki

W procesie budowania języka programowania w pierwszym kroku zostanie zbudowana jego algebra denotacji, która w jednoznaczny sposób wyznaczy odpowiadającą jej algebrę składni abstrakcyjnej. Ponieważ ta ostatnia zwykle nie jest najwygodniejsza dla użytkownika języka, przekształcamy ją do jakiejś składni konkretnej (por. rozdział 2.12) używając do tego homomorfizmu, który nie skleja „zbyt wiele” (por. twierdzenie 2.13-1). Oczywiście w podręczniku użytkownika składnia konkretna powinna zostać opisana gramatyką równaniową, rodzi się więc pytanie, czy dla każdej takiej składni istnieje odpowiadająca jej gramatyka. Aby na to pytanie odpowiedzieć trzeba wprowadzić pojęcie *funkcji szkieletowej*.

Funkcję f na słowach zbudowanych nad alfabetem A , dla której istnieje krotka słów $(S_1, \dots, S_k, S_{k+1})$ nad A taka, że

$$f.(X_1, \dots, X_k) = S_1 X_1 \dots S_k X_k S_{k+1}$$

nazwę *funkcją szkieletową* nad A , a wyznaczającą ją krotkę słów — *szkieletem* tej funkcji. Przykładem funkcji szkieletowej może być:

$$f.(wyr-b, ins_1, ins_2) = \text{if } wyr-b \text{ then } ins_1 \text{ else } ins_2 \text{ fi}$$

Szkieletem tej funkcji jest krotka słów (**if**, **then**, **else**, **fi**). Zauważmy, że funkcja

³² Sprawa komplikuje się w istotny sposób, gdy w wyrażeniach występuje kilka operacji, które nie są „łączne względem siebie”, jak. np. dodawanie i mnożenie. Każde z nich jest łączne jednakże opuszczenie nawiasów prowadzi do „niedopuszczalnej wieloznaczności”, gdyż np. $a*b+c$ jest obrazem przez K_1 dwóch wyrażeń abstrakcyjnych $*(a,+(b, c))$ oraz $+(*(a, b), c)$, których obrazy w Alg_2 będą różne. Tym problemem zajmę się w rozdziale 4.5.

$$f.(ins_1, ins_2, wyr-b) = \text{if } wyr-b \text{ then } ins_2 \text{ else } ins_1 \text{ fi}$$

nie jest funkcją szkieletową, gdyż kolejność występowania argumentów funkcji po lewej stronie równania (arność funkcji) nie odpowiada kolejności ich występowania po prawej.

W szczególnych sytuacjach funkcja szkieletowa może mieć więcej niż jeden szkielet. Na przykład dla $A = \{a\}$, funkcja $f : \{a\}^* \mapsto \{a\}^*$ określona równaniem

$$f.(x) = x \text{ a}$$

ma dwa szkielety (ϵ, a) oraz (a, ϵ) , gdyż może być w sposób równoważny zdefiniowana równaniem

$$f.(x) = a \text{ x}$$

Gdybyśmy jednak zmienili typ funkcji na $f : \{a, b\}^* \mapsto \{a, b\}^*$ pozostawiając jej równanie definicyjne niezmienionym, to miałaby ona już tylko jeden szkielet, a mianowicie (ϵ, a) .

Algebrę wielorodzajową będę nazywał *algebrą składniową*, jeżeli jest ona osiągalną algebrą słów.

Algebrę składniową będę nazywał *algebrą bezkontekstową*, jeżeli jej operacje są funkcjami szkieletowymi. Przykładami bezkontekstowych algebr składniowych są oczywiście algebry składni abstrakcyjnej. W rozdziale 2.12 pokazałem, jak dla takich algebr budować definiujące je gramatyki równaniowe. Tę konstrukcję łatwo zastosować do dowolnej algebry bezkontekstowej, co pozwala na sformułowanie następującego twierdzenia:

Twierdzenie 2.14-1 *Dla każdej algebry bezkontekstowej istnieje gramatyka równaniowa generująca jej nośniki.* ■

Jest też prawdziwe twierdzenie odwrotne:

Twierdzenie 2.14-2 *Dla każdej gramatyki równaniowej istnieje algebra bezkontekstowa, której nośnikami są języki definiowane tą gramatyką.* ■

Dowód Niech

$$X_1 = \text{wie}_1.(X_1, \dots, X_n)$$

...

$$X_n = \text{wie}_1.(X_1, \dots, X_n)$$

będzie gramatyką równaniową o rozwiązaniu $(L_1, \dots, L_n)$. Załóżmy że wielomiany tej gramatyki zostały sprowadzone do postaci sum jednomianów. Odpowiadającą jej algebrę

$$\text{Alg} = (\text{Syg}, \text{Noś}, \text{Kon}, \text{noś}, \text{kon}),$$

budujemy w sposób następujący:

- $\text{Syg} = (N_n, N_k, \text{ar}, \text{ro})$
- $N_n = \{nn_1, \dots, nn_n\}$ — jako nazwy dziedzin w naszej algebrze przyjmujemy nazwy zmiennych (zmienne) w naszej gramatyce,
- $N_k = \{nk_1, \dots, nk_m\}$ — nazwy funkcji obieramy dowolnie, ale ich liczba jest równa liczbie wystąpień jednomianów w naszej gramatyce,
- ar i ro są zdefiniowane w ten sposób, aby odpowiadały arnościom i rodzajom jednomianów naszej gramatyki,
- $\text{Noś} = \{L_1, \dots, L_n\}$,

- Kon — jest zbiorem wszystkich jednomianów naszej gramatyki,
- $\text{noś.nni} = L_i$ dla $i = 1, \dots, n$

Zauważmy teraz, że każdy jednomian naszej gramatyki jest z definicji jednomianem Chomskiego (patrz rozdział 2.5), a więc jest postaci:

$$F.(X_1, \dots, X_k) = \{S_1\} X_1 \dots \{S_k\} X_k \{S_{k+1}\}$$

gdzie S_i są słowami. Krotkę tych słów przyjmujemy jako szkielet funkcji na słowach odpowiadające jednomianowi gramatyki. Tej funkcji nadajemy jakąś nazwę nk i definiujemy ją równaniem:

$$\text{kon.nki}(x_1, \dots, x_k) = \{S_1\} x_1 \dots \{S_k\} x_k \{S_{k+1}\}$$

Na tym kończy się konstrukcja naszej algebry. Zauważmy, że wyznacza ona algebrę wynikową w sposób jednoznaczny z dokładnością do nazw nośników i konstruktorów.

Teraz należy pokazać, że nośniki Alg są zamknięte ze względu na wszystkie jej operacje oraz że Alg jest osiągalna. Ten dowód pozostawiam czytelnikowi jako pożyteczne ćwiczenie. Opisana powyżej konstrukcja algebry jest prostym odwzorowaniem opisanej w [24] konstrukcji budowania algebry dla gramatyki bezkontekstowej. ■

A oto prosty przykład wyjaśniający, jak budować algebrę dla zadanej gramatyki równaniowej. Rozważmy gramatykę równaniową dwurodzajowego języka wyrażeń:

$$\text{Liczba} = 1 \mid x \mid \text{Liczba} + \text{Liczba}$$

$$\text{Boolowska} = \text{Liczba} < \text{Liczba} \mid \text{Boolowska} \& \text{Boolowska}$$

Dla uproszczenia opuściłem nawiasy klamrowe przy nazwach funkcji. Operacje algebry odpowiadającej tej gramatyce są następujące (tu również opuszczam znaki konkatenacji), gdzie zakładam, że $w\text{-lic}$ i $w\text{-boo}$ z indeksami oznaczają odpowiednio wyrażenia liczbowe i boolowskie:

$$\text{jeden}() = 1$$

$$\text{zmienna}() = x$$

$$\text{plus}(w\text{-lic}_1, w\text{-lic}_2) = w\text{-lic}_1 + w\text{-lic}_2$$

$$\text{mniejszy}(w\text{-lic}_1, w\text{-lic}_2) = w\text{-lic}_1 < w\text{-lic}_2$$

$$\text{oraz}(w\text{-bool}_1, w\text{-bool}_2) = w\text{-bool}_1 \& w\text{-bool}_2$$

O gramatyce równaniowej powiemy, że jest *jednoznaczna* (*wieloznaczna*), gdy skonstruowana dla niej jak w dowodzie Tw. 2.14.2 algebra składniowa jest jednoznaczna (*wieloznaczna*). Intuicyjnie, gramatyka jest wieloznaczna, jeżeli istnieją takie słowa w generowanym przez nią języku, które daje się wygenerować na więcej niż jeden sposób. Te „kilka sposobów generowania”, to kilka różnych elementów składni abstrakcyjnej, które są przeciwobrazami słowa świadczącego o wieloznaczności. Na przykład słowo $1+1+1$ generowane przez naszą gramatykę, daje się uzyskać na dwa sposoby:

$$\text{plus}(1, \text{plus}(1, 1))$$

$$\text{plus}(\text{plus}(1, 1), 1)$$

Jak już zapowiadałem, składnię języka programowania będziemy budowali przez kolejne homomorficzne przekształcenia jego składni abstrakcyjnej. Ponieważ te kolejne składnie będę opisywał gramatykami równaniowymi — jako znacznie czytelniejszymi od definicji algebr —

ważne jest wiedzieć, które homomorfizmy algebr składniowych nie wyprowadzają poza algebry bezkontekstowe.

Zacznijmy od przykładu homomorfizmu, a w rzeczywistości izomorfizmu, który wyprowadza poza klasę algebr bezkontekstowych. Niech Alg będzie algebrą słów o jednym nośniku, którym jest język $\{a\}^+$ i dwóch operacjach:

$$h.() = a$$

$$f.(x) = x a$$

Jest to oczywiście algebra bezkontekstowa. Zdefiniujmy teraz algebrę podobną do niej, której jedynym nośnikiem jest język $\{a^n b^n c^n \mid n = 1, 2, \dots\}$, a operacje są zdefiniowane w sposób następujący:

$$h.() = abc$$

$$f.(a^n b^n c^n) = a^{n+1} b^{n+1} c^{n+1} \text{ dla każdego } n \geq 1.$$

Ta algebra nie jest bezkontekstowa gdyż, jak już dawno udowodniono (patrz np. [13]), jej nośnik nie daje się wygenerować gramatyką równaniową. Jednakże jest ona izomorficzna z poprzednią, a łączący je izomorfizm jest zdefiniowany równaniem:

$$l.a^n = a^n b^n c^n \text{ dla każdego } n \geq 1$$

Jak jednak łatwo zauważyć, powyższy izomorfizm nie jest funkcją szkieletową. I oczywiście nie może nią być, bo funkcja f nie jest szkieletowa.

Homomorfizm H pomiędzy dwiema algebrami składniowymi — przypominam, że taki homomorfizm, jeżeli istnieje, to jest wyznaczony jednoznacznie (Tw. 2.12-3) — nazwę *szkieletowym*, jeżeli dla każdego konstruktora $kon.nk$ pierwszej algebry, dla której:

$$ro.nk = nn$$

$$ar.nk = (nn_1, \dots, nn_n)$$

istnieje taki szkielet $(s_1, \dots, s_{n+1})$, że

$$H.nn.(kon_1.nk.(x_1, \dots, x_n)) = s_1 x_1 \dots s_n x_n s_{n+1}$$

Mówiąc w skrócie oznacza to, że homomorficznym obrazem każdej funkcji z algebry źródłowej jest funkcja szkieletowa w algebrze docelowej.

Twierdzenie 2.14-3 *Dla każdej algebry składniowej Alg następujące tezy są równoważne:*

- (1) Alg jest bezkontekstowa,
- (2) każdy homomorfizm w Alg jest homomorfizmem szkieletowym,
- (3) istnieje homomorfizm w Alg , który jest szkieletowy. ■

Czytelników zainteresowanych dowodem odsyłam do mojej pracy [24].

Rozważmy teraz prosty przykład procesu sukcesywnego budowania składni dla zadanej algebry³³. Niech będzie to bardzo prosta algebra nad liczbami o trzech operacjach:

$$\text{twórz-lic.1} : \quad \mapsto \text{Liczba}$$

$$\text{plus} : \text{Liczba} \times \text{Liczba} \mapsto \text{Liczba}$$

$$\text{razy} : \text{Liczba} \times \text{Liczba} \mapsto \text{Liczba}$$

³³ W terminach bardziej ogólnych takie procesy zostaną omówione w rozdziale 4.5.

Odpowiadająca jej składnia abstrakcyjna, oznaczmy ją $Skł-0$, jest zdefiniowana następującą gramatyką równaniową składającą się z jednego tylko równania, gdzie Wyr oznacza język wyrażeń liczbowych o wartościach stałych:

$$Wyr = \text{twórz-lc.1.}() \mid \text{plus}(Wyr, Wyr) \mid \text{razy}(Wyr, Wyr)$$

Dla uproszczenia przyjąłem te same oznaczenia, co w algebrze liczb. Pierwszym krokiem na drodze do składni ostatecznej będzie:

- zamiana $\text{twórz-lc.1.}()$ na 1,
- zamiana nazw konstruktorów plus i razy, na + i *,
- zamiana notacji prefiksowej na infiksową.

Tej zmianie odpowiada następujący homomorfizm:

$$H.[\text{twórz-lc.1.}()] = 1$$

$$H.[\text{plus}(wyr_1, wyr_2)] = (H.[wyr_1] + H.[wyr_2])$$

$$H.[\text{razy}(wyr_1, wyr_2)] = (H.[wyr_1] * H.[wyr_2])$$

Jest to oczywiście homomorfizm szkieletowy, a kolejna gramatyka równaniowa jest przez niego wyznaczona w sposób jednoznaczny:

$$Wyr = 1 \mid (Wyr + Wyr) \mid (Wyr * Wyr)$$

W drugim i ostatnim kroku procesu konstruowania składni chcielibyśmy dopuścić możliwość opuszczania „zbędnych nawiasów”, a więc na przykład pisania $1+1+1$ zamiast $(1+(1+1))$ i podobnie dla mnożenia. To jednak okazuje się niemożliwe, gdyż każdy homomorfizm usuwający nawiasy musiałby spełniać równości:

$$H.[(wyr_1 + wyr_2)] = H.[wyr_1] + H.[wyr_2]$$

$$H.[(wyr_1 * wyr_2)] = H.[wyr_1] * H.[wyr_2]$$

a to oznaczałoby sklejanie wyrażeń o różnych denotacjach, np.

$$H.[((1+1)*(1+1))] = H.(((1+(1*1))+1)) = 1+1*1+1$$

Jest to co prawda homomorfizm szkieletowy, więc odpowiadająca mu gramatyka wynikowa

$$Wyr = 1 \mid Wyr + Wyr \mid Wyr * Wyr$$

jest bezkontekstowa, jednakże odpowiadająca jej algebra jest bardziej wieloznaczna od algebry liczb, a zatem nie istnieje dla niej semantyka denotacyjna w tę algebrę.

Tradycyjny sposób, w jaki rozwiązuje się ten problem w znanych mi językach programowania — w których w ogóle zadbano o formalną definicję składni np. w Algolu [61] i Pascalu [48] — polega na przebudowaniu całego modelu języka przez wprowadzenie do algebr denotacji i składni trzech nośników $Skł$ (składnik), Czy (czynnik), Wyr (wyrażenie) wraz z następującą sygnaturą:

s-na-c	: $Skł$	$\mapsto Wyr$	<i>składnik na wyrażenie identycznościowo</i>
+	: $Skł \times Wyr$	$\mapsto Wyr$	<i>dodawanie</i>
c-na-s	: Czy	$\mapsto Skł$	<i>czynnik na składnik identycznościowo</i>
*	: $Czy \times Skł$	$\mapsto Skł$	<i>mnożenie</i>
1	:	$\mapsto Czy$	<i>generowanie 1 jako czynnika</i>
w-na-c	: Wyr	$\mapsto Czy$	<i>wyrażenie na czynnik identycznościowo</i>

Gramatyka dla składni abstrakcyjnej tego języka jest następująca:

$$\text{Wyr} = \text{s-na-l}(\text{Skł}) \mid +(\text{Skł}, \text{Wyr})$$

$$\text{Skł} = \text{c-na-s}(\text{Czy}) \mid *(\text{Czy}, \text{Skł})$$

$$\text{Czy} = 1 \mid \text{l-na-cz}(\text{Wyr})$$

a dla pierwszej (izomorficznej z nią) składni przetworzonej z zapisem infiksowym:

$$\text{Wyr} = (\text{Skł}) \mid (\text{Skł} + \text{Wyr})$$

$$\text{Skł} = (\text{Czy}) \mid (\text{Czy} * \text{Skł})$$

$$\text{Czy} = 1 \mid (\text{Wyr})$$

W tym zapisie opuściłem nazwy funkcji identycznościowych, co jednak nie niszczy jednoznaczności gramatyki, gdyż te nazwy występują w wyrażeniach należących do różnych nośników.

Teraz mogę zdefiniować szkieletowy homomorfizm usuwania nawiasów dla każdego z trzech rodzajów wyrażeń:

$$W.[(\text{skł})] = \text{skł}$$

$$W.[(\text{skł} + \text{wyr})] = S.[\text{skł}] + W.[\text{wyr}]$$

$$S.[(\text{czy})] = C.[\text{czy}]$$

$$S.[(\text{czy} * \text{skł})] = C.[\text{czy}] * S.[\text{skł}]$$

$$C.[1] = 1$$

$$C.[(\text{wyr})] = (\text{wyr})$$

który wyznacza kolejną gramatykę równaniową

$$\text{Wyr} = \text{Skł} \mid \text{Skł} + \text{Wyr}$$

$$\text{Skł} = \text{Czy} \mid \text{Czy} * \text{Skł}$$

$$\text{Czy} = 1 \mid (\text{Wyr})$$

Zapisując tę gramatykę z użyciem operacji iteracji otrzymujemy:

$$\text{Wyr} = \text{Skł} [+ \text{Skł}]^* \quad \text{wyrażenie jest sumą składników}$$

$$\text{Skł} = \text{Czy} [* \text{Czy}]^* \quad \text{składnik jest iloczynem czynników}$$

$$\text{Czy} = 1 \mid (\text{Wyr}) \quad \text{czynnik jest stałą lub wyrażeniem w nawiasach}$$

gdzie nawiasy kwadratowe są metanawiasami, a więc nie należą do składni. Tak więc w naszym języku wyrażenia są sumami składników, składniki są iloczynami czynników, a czynniki — to albo stała, albo wyrażenie ujęte w nawiasy.

Zauważmy, że zdefiniowany tu homomorfizm pozbywania się nawiasów nie jest izomorfizmem, gdyż skleja w jedno wyrażenie np. $(1+(1+1))$ oraz $((1+1)+1)$ i podobnie dla mnożenia, nie skleja jednak więcej od algebry liczb, gdyż dodawanie i mnożenie są operacjami łącznymi. Z kolei wyrażenie $((1+1)*(1+1))$ jest pozbawiane przez ten homomorfizm jedynie zewnętrznych nawiasów.

Homomorfizm denotacyjny dla odpowiadającej tej gramatyce algebry definiujemy następującymi równaniami:

$$\text{Sw}.[\text{skł}] = \text{Ss}.[\text{skł}]$$

$$\begin{aligned}
\text{Sw.}[\text{skł} + \text{wyr}] &= \text{Ss.}[\text{skł}] + \text{Sw.}[\text{wyr}] \\
\text{Ss.}[\text{czy}] &= \text{Sc.}[\text{czy}] \\
\text{Ss.}[\text{czy} * \text{skł}] &= \text{Sc.}[\text{czy}] * \text{Ss.}[\text{skł}] \\
\text{Sc.}[1] &= 1 \\
\text{Sc.}[(\text{wyr})] &= \text{Sw.}[\text{wyr}]
\end{aligned}$$

Zauważmy, że równania tego homomorfizmu wyrażają też znane ze szkoły reguły pierwszeństwa mnożenia przed dodawaniem.

Komentarz 2.14-1

Czytelnik, któremu na początku obiecałem, że denotacyjne modele języków programowania będą prowadziły do ich czytelnych definicji, może nabrać w tym miejscu wątpliwości. Jak na razie bowiem, język wyrażeń arytmetycznych — doskonale znany uczniom szkoły podstawowej — opisujemy w sposób dość zawiły i na dodatek przy użyciu zaawansowanego aparatu matematycznego. To oczywiście wymaga komentarza.

Po pierwsze, to co uczniowi podstawówki można wyjaśnić w sposób prosty i intuicyjny, komputerowi trzeba „powiedzieć” w sposób dla niego zrozumiały, a więc w języku matematyki. A właśnie jedną z zalet definicji denotacyjnych jest ich bezpośrednia przekładalność na kod kompilatora lub interpretera.

Po drugie, nasz język posłużył mi przede wszystkim jako ilustracja zastosowania metody denotacyjnej na bardzo prostym przykładzie. Jej rzeczywiste zalety będą znacznie lepiej widoczne, gdy w języku pojawią się deklaracje, instrukcje, procedury, rekurencja, obiekty i inne bardziej złożone mechanizmy programistyczne, których opis wymaga precyzyjnych narzędzi.

Po trzecie wreszcie, pisząc dla naszego języka podręcznik użytkownika możemy się śmiało odwołać do szkolnej wiedzy wyjaśniając, że wyrażenia arytmetyczne piszemy i liczymy „w zwykły sposób”, co oznacza, że w przypadku wyrażeń ani gramatyki, ani definicji denotacji w ogóle użytkownikowi nie pokazujemy. Jednakże, jak zobaczymy w rozdziale 4.5, można w tym przypadku zastosować lepsze rozwiązanie związane z pojęciem *składni kolokwialnej*.

Jakie wnioski można wyciągnąć z przedstawionego tu ćwiczenia? Wydaje się, że co najmniej dwa.

Po pierwsze, opisanie prostej, jak by się mogło wydawać, operacji opuszczania nadmiarowych nawiasów wymaga dość rozbudowanej i nie do końca oczywistej gramatyki. Jest ona niezbędna twórcy implementacji, ale z pewnością nie użytkownikowi języka, któremu wystarczy powiedzieć, że wyrażenia może pisać zgodne ze szkolnymi regułami nawiasowania.

Po drugie, pomysł, aby móc opuszczać nadmiarowe nawiasy pojawił się dopiero na poziomie drugiej algebry składniowej, gdy dwie poprzednie zostały już zbudowane. Dla jego realizacji trzeba było rozpoczynać całą pracę od nowa. W naszym prostym przykładzie nie było tej pracy aż tak wiele, ale w zastosowaniach rzeczywistych może być inaczej. Aby uniknąć takich sytuacji trzeba by myśleć o składni już na etapie algebry denotacji, co jednak burzy naszą zasadę „od denotacji do składni”. Burzy też zasadę, że denotacje powinno się budować w sposób maksymalnie naturalny.

Omawiane tu problemy były przedmiotem rozważań w pracach [22], [24] i [30], gdzie proponowano, by docelowa składnia programisty, zwana *składnią kolokwialną*, nie musiała być homomorficznym obrazem składni abstrakcyjnej. W naszym przykładzie składnia konkretna byłaby opisana gramatyką:

$$\text{Wyr} = 1 \mid (\text{Wyr} + \text{Wyr}) \mid (\text{Wyr} * \text{Wyr})$$

a składnia kolokwialna, która pozwala na opuszczanie nawiasów (choć tego nie wymusza), miałaby gramatykę:

$$\text{Wyr} = 1 \mid (\text{Wyr} + \text{Wyr}) \mid (\text{Wyr} * \text{Wyr}) \mid \text{Wyr} + \text{Wyr} \mid \text{Wyr} * \text{Wyr}.$$

Zauważmy, że algebra składni kolokwialnej nie tylko nie jest homomorficznym obrazem poprzedniej, ale nawet nie jest do niej podobna, gdyż ma inną sygnaturę. Nie istnieje więc dla niej homomorfizm denotacyjny w naszą algebrę denotacji. Z drugiej jednak strony łatwo zbudować transformację, która przekształca wyrażenia kolokwialne w konkretne przez dopisywanie do nich „brakujących” nawiasów. Tę transformację nazywam *transformacją przywracającą*.

Transformacja przywracająca nie jest oczywiście homomorfizmem, gdyż łączy dwie niepodobne do siebie algebry — a właściwiej jedynie ich nośniki — jednak za jej pomocą każdemu wyrażeniu kolokwialnemu można jednoznacznie przypisać „jego denotację” będącą denotacją odpowiadającego mu wyrażenia konkretnego. W praktyce przekładałoby się to na podręcznik użytkownika języka, który obejmowałby definicję (gramatykę) składni konkretnej, a ponadto regułę mówiącą, że nawiasy możemy opuszczać „zgodnie ze szkolnymi zasadami”³⁴.

W przypadku ogólnym w zależności od tego, na ile związek pomiędzy składnią kolokwialną i konkretną jest oczywisty, lub nie, transformację przywracającą możemy opisać w podręczniku programisty formalnie lub nieformalnie. Jej formalny opis będzie jednak niezbędny dla twórcy implementacji języka, który musi stworzyć program przekształcający programy z kolokwializmami na programy zgodne z gramatyką składni konkretnej.

Więcej na temat składni kolokwialnej piszę ogólnie w rozdziale 4.5, a kolokwializmy dla języków z serii **Lingua** są opisane w rozdziałach 5.4.3, 6.2.3, 7.8.3 i 12.9.

Na koniec jedna uwaga metodologiczna. Otóż proste języki wyrażeń, o którym była mowa w niniejszym rozdziale, obejmowały jedynie wyrażenia stałe, tj. nie zawierające zmiennych. Takie języki nie mają oczywiście praktycznego znaczenia. Ograniczyłem się do nich jedynie po to, aby pokazać związki gramatyk z algebrami na możliwie najprostszych przykładach. Pożyczając od rozdziału 4 zajmę się metodami budowania denotacyjnych modeli języków zawierających repertuar narzędzi programistycznych zbliżony do praktycznego.

³⁴ Jak zobaczymy w rozdziale 5.4.3, sytuacja może być nieco bardziej złożona.

3 Semantyczna poprawność programów

3.1 Uwagi historyczne

O semantycznej poprawności programów, zwanej historycznie *poprawnością programów* (ang. *program correctness*) mówiło się w zasadzie od czasu, gdy powstały pierwsze komputery. Najwcześniejsza praca na ten temat [66], dziś prawie całkowicie zapomniana, pochodzi o brytyjskiego matematyka Alana Turinga, który opublikował ją już w roku 1949. Znacznie później, bo w roku 1967, w zasadzie te same idee, ale opisane w sposób dalece bardziej czytelny, opublikował amerykański informatyk Richard Floyd [40]. W roku 1978 amerykańska organizacja Association of Computing Machinery ustanowiła coroczną Nagrodę Turinga za wybitne osiągnięcia w dziedzinie informatyki. Jednym z laureatów tej nagrody został w 1978 nie kto inny, jak... Richard Floyd.

Do dziś nie zostało chyba wyjaśnione, czy Floyd znał pracę Turinga. W latach 1980. pisałem w tej sprawie do Uniwersytetu w Cambridge skąd otrzymałem kategoryczną radę, abym zaprzestał prób budowania „jeszcze jednego mitu wokół Turinga”³⁵.

Praca Floyda wprowadzała bardzo ważne pojęcie *niezmiennika programu* i dotyczyła programów zapisywanych w postaci sieci działań (ang. *flow diagram*). W dwa lata później w roku 1969 brytyjski informatyk C.A.R. Hoare (później również laureat Nagrody Turinga) opublikował pracę poświęconą floydowskim ideom dla *programów strukturalnych*, tj. budowanych za pomocą konstruktorów *if-then-else* i *while*. To ujęcie, nazwane później *logiką Hoare’a*, dało asumpt do wielu dalszych badań w tym zakresie. We wczesnym okresie teorii dowodzenia poprawności programów zajmował się również Edsger W. Dijkstra [38].

Prace poświęcone poprawności programów były prowadzone także w Polsce. Jako pierwszą w tym obszarze (choć w innym matematycznie ujęciu) wymienić chyba należy publikację Antoniego Mazurkiewicza [55] z roku 1971. Rok później podczas pierwszej międzynarodowej konferencji z serii *Mathematical Foundations of Computer Science*³⁶ przedstawiliśmy —

³⁵ Alan Turing (1912-1954) jeden z twórców teorii obliczalności wprowadził uznawane za fundamentalne dla tego kierunku pojęcie, znane dziś pod nazwą *maszyny Turinga*. Dzięki pracy *O liczbach obliczalnych* w wieku 26 lat Turing został uznany za jednego z najwybitniejszych matematyków świata. Turing był też niestety przedmiotem homofobicznej dyskryminacji. W roku 1952, gdy brytyjska policja dowiedziała się, że był homoseksualistą, został postawiony przed wyborem: więzienie lub terapia hormonalna. Wybrał terapię, ale wkrótce popełnił samobójstwo.

³⁶ Tę konferencję zorganizowaliśmy wspólnie z naszymi kolegami z Wydziału Matematyki i Mechaniki Uniwersytetu Warszawskiego. Jako młodzi, nieznani jeszcze naukowcy nie mieliśmy w tamtych czasach większych szans na kontakty z nauką zachodnią. Nieznanych nikt nie kwapił się zapraszać, a z polskiej strony nie mogliśmy liczyć na sfinansowanie „dewizowych” wyjazdów. W tej sytuacji postanowiliśmy zaprosić zachód do nas. Przyjęliśmy też zasadę, aby nie zapraszać ówczesnych gwiazd nauki, które zapewne nie przyjechałyby do kraju, który z rozwoju informatyki raczej nie słynął, ale naszych rówieśników. Jak się później okazało, była to znakomita decyzja strategiczna, gdyż udało nam się zebrać przyszłą śmietankę bliskiego nam środowiska naukowego z krajów zachodnich i nie tylko. Konferencja została uznana przez uczestników za taki sukces, że nasi czechosłowaccy koledzy poprosili o zgodę na zorganizowanie konferencji pod tą samą nazwą u nich. Odbędzie się w roku 1973 w słowackich Tatrach,

Antoni Mazurkiewicz i ja — wspólną publikację [28] na zbliżony temat, która ujmowała koncepcję poprawności programów na gruncie algebry relacji, przy czym obejmowała również programy rekurencyjne i niedeterministyczne. Blisko dziesięć lat później opublikowałem pracę [20], która zawierała pełną matematyczną dyskusję reguł dowodzenia poprawności dla programów odpowiadających dowolnym sieciom działań bez procedur i rekurencji. W stosunku do innych prac w tym obszarze, a w szczególności w porównaniu z szeroko rozwijanymi wtedy ideami, w mojej pracy uwzględniałem fakt, że program może nie kończyć działania nie tylko dlatego, że wchodzi w nieskończoną pętlę, ale też i dlatego, że ulega zawieszeniu przed osiągnięciem ostatniej instrukcji.

W tym miejscu nie można też nie wspomnieć o dwóch nurtach prac rozwijanych na Uniwersytecie Warszawskim. Pierwszy, pod nazwą *logiki algorytmicznej* [8], to sformalizowane podejście do poprawności programów polegające na zbudowaniu specyficznej logiki, w której programy są składowymi formuł. W ramach drugiego nurtu [53] o znacznie bardziej inżynierskim charakterze rozwijano trzy powiązane ze sobą kierunki badawcze: *wnioskowanie gramatyczne*, *analiza wydajności systemów liczących* oraz *formalna specyfikacja wymagań*. Interesującym przykładem zastosowania trzeciego z tych kierunków jest praca [60] poświęcona badaniu poprawności oprogramowania sterującego pracą elektrowni atomowej w Darlington (Kanada).

Idea dowodzenia poprawności programów, mimo swojej niewątpliwej wartości naukowej, nie znalazła szerszych zastosowań w inżynierii oprogramowania. W moim przekonaniu stało się tak dlatego, że gdy mówimy o dowodzeniu poprawności programów milcząco zakładamy, że najpierw powstaje program, a później dowód jego poprawności. Ta kolejność jest całkowicie naturalna w matematyce — najpierw hipoteza, a później jej dowód — ale nie w inżynierii. Wyobraźmy sobie konstruktora, który najpierw buduje most, a dopiero później zabiera się do wykonania obliczeń gwarantujących, że most się nie zawali. Taki most z pewnością runąłby jeszcze w czasie budowy i tak właśnie dzieje się z programami. Pierwsza wersja kodu z reguły nie działa tak, jakby oczekiwano, poważną część budżetu na stworzenie programu poświęca się więc na tzw. uruchamianie, tj. usuwanie błędów wprowadzonych na etapie pisania kodu. Zresztą wszystkich błędów nigdy nie da się usunąć, oddaje się więc użytkownikowi program z błędami, które później są usuwane już na koszt użytkownika w ramach „serwisowania” programu.

W niniejszej książce (rozdziały 3 i 8) postanowiłem rozwinąć moje idee naszkicowane w pracach [18] i [19], gdzie proponuję wprowadzenie takich reguł budowania programów, które gwarantowałyby poprawność tych ostatnich. Zastosowanie takich reguł w procesie tworzenia programu powodowałoby, że inżynier-programista mógłby postępować tak samo, jak każdy inny inżynier, tj. budując swój produkt według reguł gwarantujących poprawność tego, co robi.

Ponieważ reguły budowania poprawnych programów powstają z reguł dowodzenia ich poprawności, rozpoczynam od dyskusji tych ostatnich. Tę dyskusję prowadzę na gruncie algebry relacji binarnych, gdyż pozwala to na proste przedstawienie idei poprawności bez wdawania się w zawiłości składni i semantyki języków programowania. Oczywiście, aby zastosować opisane niżej metody w praktyce, trzeba wprowadzić je na poziom konkretnego języka programowania, który musi mieć matematycznie opisaną semantykę i składnię. Tym więc zajmę się w rozdziałach 6 i 7, gdzie zostanie opisana pierwsza wersja języka **Lingua** by w rozdziale 8 powrócić do idei budowania programów poprawnych.

a rok później znów w Polsce. W ten sposób narodziła się seria konferencji MFCS organizowanych naprzemiennie w Polsce i Czechosłowacji aż do polskiego stanu wojennego (później wznowiona). Liczba uczestników rosła z kilkudziesięciu na początku, do kilkuset. Od roku 1974 materiały konferencyjne były wydawane przez Springer Verlag w serii Lecture Notes in Computer Science.

3.2 Programy iteracyjne

Każdy program, a także każda jego instrukcja, wyznacza pewną relację binarną (rozdz.2.6) zwaną *relacją wejście-wyjście*, która stany wejściowe programu/instrukcji wiąże ze stanami wyjściowymi. Jeżeli program jest deterministyczny, tzn. jeżeli stan początkowy wyznacza jednoznacznie stan końcowy, to ta relacja jest funkcją. Jak się okazuje, w tym abstrakcyjnym modelu daje się wyrazić całkiem sporo idei związanych z teorią poprawności programów. Od nich więc rozpoczynam mój wykład.

Niech S będzie dowolnym być może nieskończonym zbiorem obiektów, które będę nazywał *stanami*. W naszych zastosowaniach stany będą funkcjami częściowymi o skończonych dziedzinach, a więc mappingami, które identyfikatorom przypisują różnego rodzaju dane (liczby, macierze, rekordy itp.). Jednakże na poziomie ogólnym nie zakładam o zbiorze S niczego.

W najogólniejszym przypadku będę rozważał relacje binarne w zbiorze S , tj. elementy zbioru

$$\text{Rel}(S) = \{R \mid R \subset S \times S\}$$

Relacje odpowiadają programom, być może niedeterministycznym, oraz ich składowym. Fakt, że

$$a R b \quad \text{dla } a, b : S$$

oznacza, że istnieje wykonanie programu R , które rozpoczyna się w stanie początkowym a i kończy w stanie b . Brak b takiego, że $a R b$ oznacza, że próba uruchomienia programu R w stanie a prowadzi albo do jego zawieszenia, albo do niekończącego się obliczenia. W zdefiniowanym dalej języku **Lingua** będę zakładał, że każde zawieszenie się programu powoduje przejście w stan „niosący” komunikat błędu, co będzie oznaczało, że jedyną przyczyną nieokreśloności wyniku dla zadanych wartości początkowych może być nieskończone obliczenie. Na poziomie ogólnym nie czynię jednak takiego założenia.

W czasach informatycznej prehistorii, a więc na przełomie lat 1940/1950, programy były listami etykietowanych instrukcji, które wykonywano w kolejności ich występowania w programie, chyba że pojawiła się instrukcja **goto**, zwana *instrukcją skoku*, która zaburzała ten porządek. Za pomocą tej instrukcji oraz konstruktora warunkowego **if-then** można było zbudować dowolnie złożony graf instrukcji, zwany *siecią działań*, lub też z angielska — flow-diagrammem. Pierwsze prace związane z dowodzeniem poprawności (por. [38] i [55]) dotyczyły więc takich właśnie programów nazwanych później *programami iteracyjnymi*.

Najogólniejszym modelem relacyjnym dla programu iteracyjnego jest układ równań stałopunktowych

$$\begin{aligned} X_1 &= R_{11} X_1 \mid \dots \mid R_{1n} X_n \mid R_1 \\ &\dots \\ X_n &= R_{n1} X_1 \mid \dots \mid R_{nn} X_n \mid R_n \end{aligned} \tag{3.2-1}$$

odpowiadający grafowi, którego węzłami są liczby $1, \dots, n$, a relacje R_{ij} są przypisane krawędziom. Ze względu na fakt, że współczynniki zmiennych stoją po ich lewych stronach, ten typ równań nazwano *lewostronnie liniowymi*. Kod takiego programu może być zapisany w postaci sekwencji instrukcji etykietowanych postaci:

$$i : \text{do } R_{ij} \text{ goto } j.$$

Zauważmy, że ponieważ R_{ij} mogą być dowolnymi relacjami, każda taka instrukcja, a także program, na który się one składają, ma charakter niedeterministyczny. Aby układ (3.2-1) reprezentował program deterministyczny muszą być spełnione dwa warunki:

- wszystkie R_{ij} są funkcjami,
- dla każdego i , wszystkie $R_{i1}, \dots, R_{in}$ mają rozłączne dziedziny.

Jeżeli przez $(P_1, \dots, P_n)$ oznaczymy rozwiązanie równania (3.2-1), to P_i jest relacją wejście-wyjście na drodze od wierzchołka 1 do i . Jeżeli więc przyjmiemy, że 1 jest wierzchołkiem początkowym, a n — a końcowym, to P_n jest relacją wynikową programu. Tak rozumiana klasa programów iteracyjnych wraz z odpowiadającymi im regułami dowodzenia całkowitej poprawności została szczegółowo opisana w [20]³⁷.

Programiści przełomu lat 1950/60 prześcigali się w budowaniu coraz bardziej wymyślnych programów, szcząc się nierzadko tym, że nikt poza autorem programu nie potrafił zrozumieć jego działania. Niestety zdarzało się też dość często, że nawet autor nie był w stanie przewidzieć do końca, jak zachowa się program w jakichś nietypowych sytuacjach.

W reakcji na te zjawiska zaczęły powstawać pierwsze algorytmiczne języki programowania takie jak Fortran i Algol, które oferowały narzędzia tzw. *programowania strukturalnego*³⁸, pozwalającego na eliminowanie *goto* przez użycie konstruktorów pętli typu *while* lub *for*. Każdy zbudowany w ten sposób program dawał się rozłożyć na instrukcje elementarne połączone ze sobą konstruktorami następstwa, rozgałęzienia i pętli. Takie programy były znacznie łatwiejsze do zrozumienia, a także pozwalały na istotne uproszczenie reguł dowodzenia poprawności. Wystarczyło bowiem dla każdego konstruktora programistycznego zbudować odpowiadającą mu regułę, która by z poprawności jego argumentów pozawalała wnioskować o poprawności wyniku.

W dalszym ciągu ograniczę się do trzech *konstruktorów strukturalnych*, gdyż pozostałe (np. *for*) mogą być łatwo zdefiniowane za ich pomocą:

1. konstruktor sekwencyjnego łączenia instrukcji oznaczany symbolem średnika „;”,
2. konstruktor warunkowy *if-then-else-fi*,
3. konstruktor pętli iteracyjnej *while-do-od*.

Złożeniu sekwencyjnemu odpowiada oczywiście złożenie relacji (funkcji). Dla zdefiniowania dwóch pozostałych, muszę wprowadzić dodatkowe pojęcia związane z faktem, że występujące w nich formuły logiczne, czyli predykaty, najczęściej nie są klasycznymi predykatami dwuwartościowymi (por. rozdział 2.9).

Zacznijmy od obserwacji, że każdy predykat p na stanach jest jednoznacznie reprezentowany przez parę dwóch rozłącznych zbiorów stanów:

$$C = \{s \mid p.s = pp\}$$

$$\neg C = \{s \mid p.s = ff\}$$

Jeżeli p jest klasycznym predykatem dwuwartościowym, to oczywiście $C \mid \neg C = S$, ale w typowych sytuacjach programistycznych tak już nie jest. Przypominam, że trzecia wartość

³⁷ Analogicznie do rekurencji lewostronnie liniowej można też mówić o rekurencji prawostronnie liniowej odpowiadającej równaniom typu $X = XR \mid Q$. Również one są omówione w [20].

³⁸ Autorem, który wprowadził do literatury pojęcie programowania strukturalnego był holenderski informatyk Edsger Dijkstra [37] i [38].

logiczna w rachunku McCarthy'ego odpowiada zarówno sytuacji, w której wartością wyrażenia boolowskiego jest błąd abstrakcyjny, jak i tej, w której proces obliczania wartości wyrażenia się nie kończy.

Niech teraz relacje P i Q reprezentują dowolne programy, a para rozłącznych zbiorów stanów $(C, \neg C)$ — dowolny predykat trójwartościowy. Wskazane wyżej trzy konstruktory programistyczne definiuję w następujący sposób:

1. $P ; Q \quad \quad \quad = P Q$
2. **if** $(C, \neg C)$ **then** P **else** Q **fi** $\quad = [C] P \mid [\neg C] Q$
3. **while** $(C, \neg C)$ **do** P **od** $\quad = ([C] P)^* [\neg C]$

gdzie $[C]$ oznacza relację identyczności na C (patrz rozdział 2.6). Ten trzeci konstruktor można zdefiniować również jako rozwiązanie równania stałopunktowego (por. Tw. 3.2-1)

$$X = [C] P X \mid [\neg C]$$

Zauważmy, że i dwa pozostałe konstruktory można traktować jako rozwiązania równań stałopunktowych, choć trywialnych, bo nie zawierających zmiennej po prawej stronie:

$$X = P Q$$

$$X = [C] P \mid [\neg C] Q$$

3.3 Procedury i rekurencja

Kolejnym ważnym krokiem na drodze do rozwoju technik programowania było wprowadzenie procedur, a w tym procedur rekurencyjnych. Najogólniej rzecz ujmując procedura to relacja:

$$P : \text{Rel}(S)$$

której nadano nazwę pozwalającą na uruchamianie procedury w różnych miejscach programu. W językach proceduralnych pojawiły się więc dwie nowe konstrukcje:

- deklaracje procedury,
- instrukcje wywołania procedury.

Te pierwsze stanowiły nowy typ obiektu, te drugie były jedynie nowym typem instrukcji.

Procedury deklarowano przez zadanie pewnej makroinstrukcji, zwanej *treścią procedury*, uzupełnionej o mechanizmy przekazywania i zwracania parametrów, które na tym etapie ogólności pominię, gdyż można je traktować jako transformacje stanów w stany, a więc składowe treści występujące na początku i na końcu.

Skoro wśród występujących w programie instrukcji mogły pojawiać się wywołania procedur, to mogły się też pojawiać w ich treściach. Innymi słowy procedury mogły wywoływać w swoich treściach inne procedury, co było ważnym krokiem na drodze do programowania strukturalnego. Początkowo jednakże nie godzono się na to, aby procedury wywoływały same siebie, czy to bezpośrednio w swojej treści, czy też pośrednio, tj. w treści wywoływanej przez nie innej procedury. Tak właśnie było we wczesnych wersjach języka Fortran, a także w zbudowanym w Zakładzie Aparatów Matematycznych PAN języku SAKO, o którym piszę nieco więcej w rozdziale 7.1.

Możliwość wywoływania procedury przez nią samą wprowadzono po raz pierwszy do powstałego na początku lat 1960. języka Algol 60³⁹ [61], a tak tworzone procedury nazwano *procedurami rekurencyjnymi*. Dekadę później pojawiły się one również w języku Pascal [48].

Na gruncie algebry relacji wprowadzenie procedur rekurencyjnych oznacza, że jako konstruktory programów dopuszcza się obok równań lewostronnie liniowych postaci (3.2-1), również *nieliniowe równania wielomianowe* postaci:

$$X_1 = \Psi_1(X_1, \dots, X_n)$$

...

$$X_n = \Psi_n(X_1, \dots, X_n)$$

gdzie każda z funkcji $\Psi_i(X_1, \dots, X_n)$ jest zbudowana jako kombinacja zmiennych i stałych połączonych operacjami składania i sumowania relacji. Takie układy równań można potraktować jako pojedyncze równania stałopunktowe w łańcuchowo zupełnym zbiorze wektorów relacji uporządkowanym po współrzędnych (por. rozdział 2.3):

$$\text{Rel}(S)^{\text{cn}} = \{(R_1, \dots, R_n) \mid R_i : \text{Rel}(S)\}$$

Każdy taki układ równań wyznacza więc funkcję wektorową

$$\Psi : \text{Rel}(S)^{\text{cn}} \mapsto \text{Rel}(S)^{\text{cn}}$$

$$\Psi.(R_1, \dots, R_n) = (\Psi_1.(R_1, \dots, R_n), \dots, \Psi_n.(R_1, \dots, R_n))$$

Jeżeli wszystkie Ψ_i są ciągłe względem wszystkich swoich zmiennych, to ciągła jest również funkcja wektorowa Ψ . Dla takich funkcji prawdziwe jest więc ogólne twierdzenie Kleene'ego (rozdział 2.3).

O ile zagadnienia poprawności programów iteracyjnych były przedmiotem licznych badań naukowych w latach 1960-1980, o tyle poprawnością programów z procedurami rekurencyjnymi zajmowano się niewiele. W moim przekonaniu główną przyczyną tego stanu rzeczy był fakt, że w przypadku procedur nie pomyślano o wprowadzeniu analogicznych jak dla programów iteracyjnych konstruktorów strukturalnych. Dla zbadania tej możliwości rozważę w rozdziałach 3.5.2 i 3.6.2 prosty schemat procedury rekurencyjnej z jednym wywołaniem odpowiadającej równaniu:

$$X = \text{HXT} \mid E \tag{3.3-2}$$

gdzie $H, T, E : \text{Rel}(S)$ są relacjami, które nazywam odpowiednio *głową procedury* (ang. head), *ogonem procedury* (ang. tail) i *wyjściem procedury* (ang. exit). Ograniczoność tego schematu polega na tym, że procedura rekurencyjna wywołuje samą siebie jedynie bezpośrednio i tylko jednokrotnie. W dalszym ciągu ten schemat będę nazywał *rekurencją prostą*.

Zauważmy, że równanie (3.3-2) obejmuje przypadek iteracyjnej instrukcji **while**, gdy $H = [C]P$, $T = [S]$, $E = [\neg C]$

Na koniec jedna uwaga metodologiczna. W budowanym później języku programowania **Lingua**, programy mają charakter deterministyczny, a więc odpowiadają funkcjom, a nie relacjom. Jednakże w wykładzie ogólnej teorii poprawności programów do przypadku

³⁹ Ten język nazywano też w skrócie Algolem, jednakże nazwa Algol 60 jest o tyle uzasadniona, że później próbowano zbudować uniwersalny język programowania wszechczasów, który nazwano Algol 68. Budowa tego języka zakończyła się na poziomie definicji, nigdy jednak nie doczekał się on pełnej implementacji. Podobnie też zakończyły się próby tworzenia innych języków uniwersalnych, takich jak Ada, czy Delphi. Te ostatnie miały swoje implementacje, nie zdobyły jednak większego uznania w świecie programistów i w rezultacie zostały zarzucone.

deterministycznego ograniczam się jedynie budując regułę całkowitej poprawności pętli **while**, gdyż poza nią determinizm nie prowadzi do istotnego uproszczenia reguł.

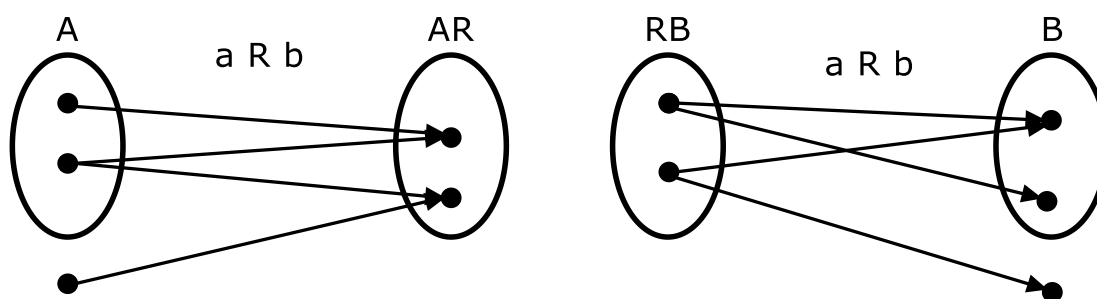
3.4 Dwa pojęcia poprawności programów

Dla wyrażenia pojęcia poprawności programów na gruncie algebry relacji binarnych zdefiniuję dwie operacje składania zbioru z relacją. Obie przypominają zdefiniowaną w rozdziale 2.6 operację sekwencyjnego składania relacji. W dalszym ciągu przez $A, B, C, \dots$ będę oznaczał podzbiory zbioru S , a przez $P, Q, R, \dots$ — relacje należące do $\text{Rel}(S)$. Obie operacje składania oznaczam tym samym symbolem, co składanie relacji:

$A \bullet R = \{b \mid (\exists a:A) a R b\}$ — *złożenie lewostronne*; obraz A względem R

$R \bullet B = \{a \mid (\exists b:B) a R b\}$ — *złożenie prawostronne*; przeciwobraz B względem R .

W przyszłości znak złożenia będzie najczęściej opuszczany, będziemy więc pisać AR oraz RA .



Rys. 3.4-1 Lewo- i prawostronne złożenie zbioru z relacją

Wyrażając to inaczej:

AR oznacza zbiór stanów końcowych dla wszystkich wykonań R zaczynających się w A , ale ze względu na niedeterminizm R niektóre z nich mogą być równocześnie stanami końcowymi wykonań zaczynającymi się poza A ,

RB oznacza zbiór stanów początkowych, dla których istnieją wykonania programu R kończące się w B , ale też, ze względu na niedeterminizm R , mogą jednocześnie istnieć wykonania rozpoczynające się w tych stanach i kończące się poza B .

Operacje składania zbioru z relacją mają własności podobne do operacji składania relacji. Na przykład są lewo- i prawostronnie łączne, tj.

$$A(RQ) = (AR)Q$$

$$(RQ)B = R(QB)$$

oraz lewo- i prawostronnie rozdzielne względem sum zbiorów i relacji:

$$(A \mid B) R = (AR) \mid (BR)$$

$$A (R \mid Q) = (AR) \mid (AQ)$$

...

Są też monotoniczne względem każdego argumentu:

jeżeli $A \subset B$ to $AR \subset BR$

jeżeli $R \subset Q$ to $AR \subset AQ$

i analogicznie dla składania prawostronnego. W rzeczywistości operacja składania jest też ciągłą względem każdego z argumentów z osobna w sensie zdefiniowanym w rozdz.2.3. W dalszej części będę zakładał, że składanie wiąże silniej od sumy, co pozwoli mi pisać:

$AR \mid BR$ zamiast $(AR) \mid (BR)$

Przypominam też (rozdział 2.6), że

$[A] = \{(a, a) \mid a:A\}$

oznacza podzbiór relacji identyczności (a więc funkcji) na stanach ograniczony do zbioru A .

Lemat 3.4-1 Dla dowolnych $A, B, C \subseteq S$ oraz $R : \text{Rel}(S)$ są prawdziwe następujące zależności:

- (1) $[A]B = A \cap B$
- (2) $A[B] = A \cap B$
- (3) $(A \cap B)R = A[B]R$
- (4) $R(A \cap B) = R[A]B$
- (5) $(A \cap B)R \subseteq C$ jest równoważne $A[B]R \subseteq C$
- (6) jeżeli $A \subseteq [B]RC$ to $(A \cap B) \subseteq RC$ ■

Sprawdzenie tych zależności pozostawiam Czytelnikowi.

Teraz jesteśmy przygotowani do zdefiniowania dwóch podstawowych pojęć związanych z poprawnością programów: *poprawności częściowej* i *poprawności całkowitej*. W obu przypadkach chodzi o wyrażenie faktu, że jeżeli dane wejściowe spełniają pewne warunki, to dane wyjściowe mają oczekiwane własności. Na przykład, od programu sortującego listy oczekujemy, aby dla każdej odpowiednio skonfigurowanej listy na wejściu (warunek początkowy) program oddał na wyjściu listę posortowaną (warunek końcowy).

Ponieważ każdej własności stanów jednoznacznie odpowiada zbiór stanów o danej własności, poprawność programu P względem zbioru stanów początkowych A zwanego *prewarunkiem* i oczekiwanych końcowych B zwanego *postwarunkiem* można bardzo prosto wyrazić w algebrze zbiorów i relacji:

$AP \subseteq B$ — *poprawność częściowa* względem prewarunku A i postwarunku B

$A \subseteq PB$ — *poprawność całkowita* względem prewarunku A i postwarunku B

Poprawność częściowa oznacza, że każde obliczenie, które zaczyna się w A , jeżeli się kończy, to kończy się w B . Poprawność częściową programu zapisujemy formułą:

$[\text{ParPre } A] P [\text{ParPost } B]$

Warunek A nazywamy *częściowym prewarunkiem* (ang. *partial precondition*), a warunek B — *częściowym postwarunkiem* (ang. *partial postcondition*).

Poprawność całkowita oznacza, że dla każdego stanu początkowego w A , istnieje obliczenie kończące się w B . Poprawność całkowitą programu zapisujemy formułą:

$[\text{TotPre } A] P [\text{TotPost } B]$

Warunek A nazywamy *całkowitym prewarunkiem* (ang. *total precondition*), a warunek B — *całkowitym postwarunkiem* (ang. *total postcondition*).

Zauważmy teraz, że najmniejszym zbiorem B dla którego zachodzi $AP \subseteq B$ jest oczywiście AP . Ten zbiór nazywamy więc *najsilniejszym postwarunkiem częściowym* dla prewarunku A i

programu P . Jest to więc najsilniejszy postwarunek jakiego możemy oczekiwać od stanów końcowych wykonań rozpoczynających się w A .

Analogicznie PB nazwiemy *najsłabszym prewarunkiem całkowitym* dla postwarunku B i programu P . Jest to więcajsłabszy prewarunek gwarantujący, że P zostanie wykonane i przynajmniej jedno z wykonań zakończy się w B .

Definicje częściowej i całkowitej poprawności zapisane za pomocą kwantyfikatorów wyglądają następująco:

$$\begin{aligned}
 AP \subset B & \quad \text{—} \quad (\forall a:A) (\forall b:B) \text{ jeżeli } aPb \text{ to } b:B \\
 & \quad \text{każde obliczenie } P, \text{ które zaczyna się od } a:A, \text{ jeżeli się kończy, to kończy się w } B, \text{ ale być może takie obliczenie nie istnieje,} \\
 A \subset PB & \quad \text{—} \quad (\forall a:A) (\exists b:B) aPb \\
 & \quad \text{dla każdego } a:A \text{ istnieje obliczenie } P, \text{ które zaczyna się w } a \text{ i kończy w } B, \text{ ale mogą też istnieć obliczenia zaczynające się w } a \text{ i nie kończące się w } B.
 \end{aligned}$$

Jak więc widzimy, w przypadku ogólnym programów niedeterministycznych żadna z tych własności nie jest silniejsza od drugiej. Jednakże dla programów deterministycznych (tj. gdy P jest funkcją) poprawność całkowita oznacza, że dla każdego $a:A$ to (jedyne) obliczenie, które zaczyna się w A , kończy się w B , a więc każde obliczenie, które zaczyna się w A kończy się w B . Tak więc, jeżeli F jest funkcją, to poprawność całkowita implikuje częściową:

$$\text{jeżeli } A \subset FB \text{ to } AF \subset B \quad (3.4-1)$$

Jest też prawdziwa następująca implikacja:

$$\text{jeżeli } AF \subset B \text{ oraz dla każdego } a:A, F.a \text{ jest określone, to } A \subset FB \quad (3.4-2)$$

Innymi słowy, jeżeli F jest funkcją całkowitą na A , to poprawność częściowa implikuje całkowitą. Rzeczywiście, niech $a:A$. Wtedy istnieje b takie, że $b = F.a$. Skoro jednak $AF \subset B$, to $b:B$, a więc $a:FB$. W ten sposób udowodniliśmy następujące twierdzenie:

Twierdzenie 3.4-1 Program deterministyczny F jest całkowicie poprawny względem A i B wtedy i tylko wtedy, gdy jest częściowo poprawny względem A i B oraz dla każdego $a:A$ jego stan końcowy jest określony. ■

Jak stąd wynika, dowód całkowitej poprawności programu deterministycznego F możemy zawsze podzielić na dwa etapy:

- I. dowód poprawności częściowej, tj. $AF \subset B$,
- II. dowód określoności F na całym A , tj. $A \subset FS$

Zauważmy teraz, że jeżeli w języku programowania wprowadzimy mechanizm błędów abstrakcyjnych, to za pomocą poprawności częściowej możemy wyrazić fakt, że przy zadanych warunkach początkowych żaden stan końcowy (jeżeli istnieje!) nie niesie komunikatu błędu, a więc że program nie ulegnie zawieszeniu. W takim przypadku jedyna sytuacja, gdzie $F.a$ może być nieokreślone, ma miejsce wtedy, gdy obliczenie o początku w a się nie kończy.

Jeżeli $F.a$ jest określone, to mówimy, że program reprezentowany przez F ma w stanie a *własność stopu*.

Jak pokazuje praktyka, własność stopu wielu programów jest tak oczywista, że dowód w zasadzie można pominąć. Na przykład, gdy jedynymi składowymi programu, które mogłyby generować nieskończone obliczenia są pętle typu **while**, które przeszukują skończone zbiory danych, to po skończonej liczbie kroków, program musi się zatrzymać.

Warto jednak wiedzieć, że istnieją programy, dla których udowodnienie własności stopu może być niezwykle trudne. Jeden z takich przykładów umieszczono na elewacji budynku Biblioteki Uniwersytetu Warszawskiego. Hipoteza o całkowitej poprawności tego programu wygląda tak:

```
TotPre  n > 0
      x := n;
      while x > 1 do;
        if x mod 2 = 0 then x := x/2 else x := 3x + 1
      TotPost x = 1
```

Jest to sprowadzona do problemu stopu tzw. *hipoteza Collatza* sformułowana w roku 1937 i do tej pory nie rozstrzygnięta, mimo, że zajmowali się nią różni wielcy matematycy, a w tym i Stanisław Ulam. Myślę też, że wielu studentów matematyki i informatyki uwiodła pozorna prostota tej hipotezy⁴⁰. Na razie wiadomo jedynie tyle, że hipoteza jest prawdziwa dla $n < 5 \cdot 2^{60}$.

Podobnie było z Wielkim Twierdzeniem Fermata⁴¹, które również można sprowadzić do problemu stopu. Ogłoszone w roku 1637 zostało udowodnione dopiero w roku 1994 przez angielskiego matematyka Andrew Wilesa. Jego dowód obejmował 100 stron i był przeprowadzony na gruncie topologicznej teorii krzywych eliptycznych.

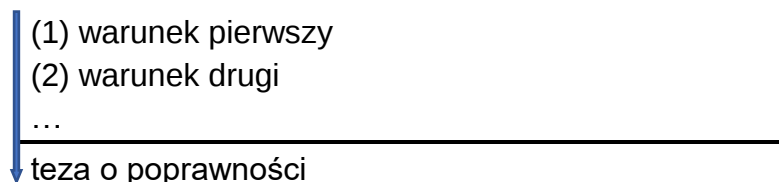
Na gruncie ogólnej teorii obliczalności udowodniono (Alan Turing), że nie da się zbudować algorytmu, który dla każdego programu i każdego jego stanu początkowego rozstrzygałby w skończonej liczbie kroków, czy program zakończy działanie.

Twierdzenie 3.4-2 *W ogólnym przypadku problem stopu programu jest nierozstrzygalny.*

W dalszej części rozważań nad poprawnością programów reguły dowodzenia poprawności będą wyrażane przez wskazywanie w jaki sposób z poprawności składowych programu można wnioskować o poprawności całego programu. W najogólniejszej postaci te reguły będę zapisywał za pomocą następującego schematu:

⁴⁰ Sam padłem kiedyś jej ofiarą, gdy referując własną pracę na temat całkowitej poprawności programów na uniwersytecie w Saarbrücken stwierdziłem, że moją metodą można łatwo udowodnić zatrzymanie się programu. Gdy to powiedziałem, ktoś z sali poprosił, abym zilustrował to na przykładzie, który mi zaraz poda, po czym podał program Collatza. Nie znałem tego przykładu, więc zapisałem program na tablicy i przystąpiłem do jego analizy. Nie wychodziło mi jednak — co brałem na karb zdenerwowania — więc oświadczyłem, że pomyślę nad tym wieczorem. Wieczorem jednak nadal nie miałem tego dowodu. Co za wstyd — taki prosty program, a nie potrafię sobie z nim poradzić. Po powrocie do Warszawy pokazałem go kolegom i wtedy zostałem oświecony, że nie ja jeden nie rozstrzygnąłem hipotezy Collatza. Prawdziwie mi ulżyło.

⁴¹ Twierdzenie to orzeka, że dla żadnej liczby naturalnej $n > 2$ nie istnieją takie liczby naturalne dodatnie x , y i z , które spełniałyby równanie $x^n + y^n = z^n$. To twierdzenie Fermat zapisał na marginesie jednej z czytanych książek dodając, że znalazł zadziwiająco prosty dowód twierdzenia, jednak margines jest za mały, aby go mógł pomieścić. Fermat sformułował w tym trybie niejedno twierdzenie bez dowodu, dla których inni matematycy dowody znajdowali stosunkowo szybko. Jednakże jego najstojniejszego twierdzenie czekało na tę chwilę ponad 350 lat, a znaleziony dowód okazał się bardzo daleki od prostego.



gdzie strzałka wskazuje kierunek wynikania. W niektórych regułach będą występowały strzałki dwustronne wskazujące na wynikanie w obu kierunkach. Jak przekonamy się nieco dalej, strzałka skierowana w dół wskazuje, jak zbudować program poprawny z jego poprawnych składowych.

3.5 Poprawność częściowa

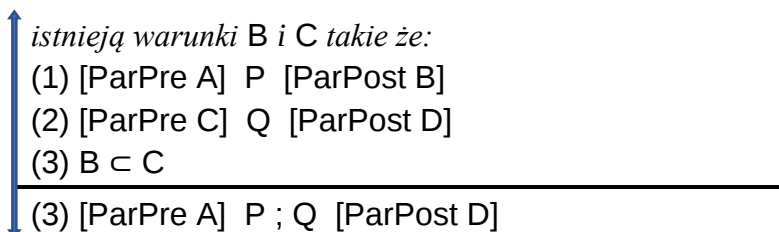
Przy definiowaniu reguł dowodzenia poprawności programów warto wyróżnić dwie klasy konstruktorów programów: *konstruktory proste* nie wprowadzające do programu mechanizmu powtarzalności oraz konstruktory wprowadzające powtarzalność, które nazwę *rekurencyjnymi*. Te pierwsze definiujemy przez proste kombinacje operacji złożenia i sumy relacji, te drugie wymagają użycia równań stałopunktowych. Z tego punktu widzenia można powiedzieć, że iteracja jest szczególnym rodzajem rekurencji

3.5.1 Złożenie i rozgałęzienie

Najczęściej spotykanymi konstruktorami prostymi są złożenie sekwencyjne i rozgałęzienie programów, do tych więc konstruktorów ograniczę reguły dowodzenia poprawności programów prostych.

Reguła 3.5.1-1 Częściowa poprawność złożenia

Dla dowolnych $A, D \subset S$ oraz $P, Q : \text{Rel}(S)$ jest spełniona reguła:



Dowód. Założenia nad kreską wyrażone algebraicznie mają postać:

$$AP \subset B$$

$$CQ \subset D$$

Stąd i z monotoniczności składania

$$(AP) \ Q \subset CQ \subset D$$

z więc z łączności składania

$$A \ (PQ) \subset D$$

Dla udowodnienia wynikania w drugą stronę przyjmujemy

$$B = C = AP$$

Stąd $AP \subset B$ oraz $BQ = APQ \subset D$ ■

Reguła 3.5.1-2 Częściowa poprawność if-then-else

Dla dowolnych $A, D, C, \neg C \in S$ oraz $P, Q : \text{Rel}(S)$, jeżeli $C \cap \neg C = \emptyset$, to jest spełniona reguła:

$$\frac{\begin{array}{l} (1) [\text{ParPre } A \cap C] \quad P \quad [\text{ParPost } B] \\ (2) [\text{ParPre } A \cap \neg C] \quad Q \quad [\text{ParPost } B] \end{array}}{[\text{ParPre } A] \quad \text{if } (C, \neg C) \text{ then } P \text{ else } Q \text{ fi } \quad [\text{ParPost } B]}$$

Dowód. Zauważmy, że (1) i (2) można zapisać jako:

$$A [C] P \subset B$$

$$A [\neg C] Q \subset B$$

a dodając te zawierania stronami otrzymujemy

$$A ([C] P \mid [\neg C] Q) \subset B \quad \blacksquare$$

Na koniec jeszcze trzy dość oczywiste reguły wynikające wprost z monotoniczności operacji składania zbioru z relacją.

Reguła 3.5.1-3 Wzmocnienie częściowego prewarunku

Dla każdej $P : \text{Rel}(S)$ oraz każdych $A, B, C \in S$ jest spełniona następująca reguła:

$$\frac{\begin{array}{l} [\text{ParPre } A] \quad P \quad [\text{ParPost } B] \\ C \subset A \end{array}}{[\text{ParPre } C] \quad P \quad [\text{ParPost } B]}$$

Reguła 3.5.1-4 Osłabienie częściowego postwarunku

Dla każdej $P : \text{Rel}(S)$ oraz każdych $A, B, C \in S$ jest spełniona następująca reguła:

$$\frac{\begin{array}{l} [\text{ParPre } A] \quad P \quad [\text{ParPost } B] \\ B \subset C \end{array}}{[\text{ParPre } A] \quad P \quad [\text{ParPost } C]}$$

Reguła 3.5.1-5 Koniunkcja pre- i postwarunków

Dla każdej $P : \text{Rel}(S)$ oraz każdych $A, B, C, D \in S$ jest spełniona następująca reguła:

$$\frac{\begin{array}{l} [\text{ParPre } A] \quad P \quad [\text{ParPost } B] \\ [\text{ParPre } C] \quad P \quad [\text{ParPost } D] \end{array}}{[\text{ParPre } A \cap C] \quad P \quad [\text{ParPost } B \cap D]}$$

Jak więc widzimy dowód częściowej poprawności programu strukturalnego bez rekurencji sprowadza się do trzech zadań:

- A. wskazania wymaganych pre- i postwarunków pośrednich,
- B. udowodnienia częściowej poprawności składowych programu,

C. udowodnienia implikacji pomiędzy odpowiednimi pre- i postwarunkami.

Występujące w dowodach pre- i postwarunki nazywamy *asercjami*.

Główna trudność przy dowodzeniu poprawności programu leży we wskazywaniu asercji niezbędnych do przeprowadzenia dowodu. Może to być zarówno trudność o charakterze matematycznym — jakie własności powinny wyrażać⁴² — ale też i praktycznym: jak w czytelny sposób wyrazić warunki obejmujące najczęściej duże zbiory występujących w programie zmiennych. Do tych problemów wrócę w rozdziale 8. W tym miejscu warto też podkreślić, że wpisywane do programu asercje mogą grać rolę wewnętrznych specyfikacji programu użytecznych przy jego dalszej obróbce programistycznej.

W niniejszym rozdziale pominąłem całkowicie problem dowodzenia poprawności składowych atomowych programów, takich jak np. instrukcja przypisania i deklaracje zmiennych, gdyż na obecnym poziomie ogólności (dowolne funkcje na abstrakcyjnych stanach) niczego praktycznego nie da się na ten temat powiedzieć. To zagadnienie pozostawiam więc do czasu, gdy reguły dowodzenia, a właściwie już wtedy reguły budowania programów, będę tworzyć dla konkretnego języka programowania (rozdział 8).

3.5.2 Rekurencja i iteracja

Dla sformułowania reguł dowodzenia poprawności procedur wzajemnie rekurencyjnych uogólniam operacje składania relacji oraz składania zbioru z relacją na przypadek wektorowy:

$$(P_1, \dots, P_n) (R_1, \dots, R_n) = (P_1 R_1, \dots, P_n R_n)$$

i analogicznie dla obu złożań zbioru z relacją. Uogólniam też na przypadek wektorowy relację zawierania się zbiorów, tak jak czyniłem to w przypadku wektorowych zbiorów łańcuchowo zupełnych w rozdziale 2.3. Tak więc:

$$(A_1, \dots, A_n) \subset (B_1, \dots, B_n) \text{ oznacza } A_1 \subset B_1 \text{ oraz } \dots \text{ oraz } A_n \subset B_n$$

Dla uproszczenia zapisu wektorowe zawieranie oznaczam tym samym symbolem, co zawieranie zwykłe. W dalszym ciągu pismem pogrubionym będę oznaczał wektory relacji i zbiorów, a także funkcje na takich obiektach.

O wektorze relacji **R** powiem, że jest *częściowo poprawny* względem wektorów zbiorów **A** i **B** (o wspólnymi z nim liczbami elementów), gdy **A R** **B** co zapisuję jako

$$[\text{ParPre } \mathbf{A}] \mathbf{R} [\text{ParPost } \mathbf{B}]$$

i analogicznie dla *poprawności całkowitej*. Pojęcie funkcji ciągłej przenosi się na funkcje wektorowe w sposób oczywisty.

Teraz możemy sformułować regułę dowodzenia częściowej poprawności dla najogólniejszego przypadku wektora relacji będącego punktem stałym funkcji ciągłej. Ten przypadek obejmuje oczywiście funkcje wielomianowe, ponieważ jednak dla ogólnego przypadku wielomianów nie potrafię napisać lepszej reguły niż dla dowolnych funkcji ciągłych, pozostają przy tych ostatnich. Takie reguły mogą być jednak sformułowane dla konkretnych prostych wielomianów, co jest pokazane nieco dalej w tym rozdziale.

⁴² To że do tej pory nie udało się udowodnić hipotezy Collatza (rozdział 3.4) oznacza właśnie, że nie udało się znaleźć asercji, która byłaby niezmiennikiem pętli opisującej ten problem. O niezmiennikach pętli w rozdziałach 3.5.2 i 3.6.2.

Reguła 3.5.2-1 Częściowa poprawność wektora relacji zdefiniowanego stałopunktowo

Dla każdej wektorowej funkcji ciągłej $\Psi : \text{Rel}(S)^{cn} \mapsto \text{Rel}(S)^{cn}$, jeżeli R jest najmniejszym rozwiązaniem równania $X = \Psi.X$, to dla każdych $A, B : S^{cn}$ jest spełniona następująca reguła, gdzie $\emptyset = (\emptyset, \dots, \emptyset)$ jest n -elementowym wektorem relacji pustych:

$$\begin{array}{l} \uparrow \text{istnieje rodzina wektorów warunków } \{B_i \mid i \geq 0\} \text{ taka, że} \\ (1) (\forall i \geq 0) [\text{ParPre } A] \Psi^i.\emptyset [\text{ParPost } B_i] \\ (2) U\{B_i \mid i \geq 0\} \subset B \\ \hline (3) [\text{ParPre } A] R [\text{ParPost } B] \end{array}$$

Dowód. Z twierdzenia Kleene'ego (rozdz.2.3)

$$R = U\{\Psi^i.\emptyset \mid i \geq 0\}$$

Dodając stronami zawierania (1) otrzymujemy

$$A U\{\Psi^i.\emptyset \mid i \geq 0\} \subset U\{B_i \mid i \geq 0\}$$

a więc po uwzględnieniu (2) otrzymujemy (3). Dla udowodnienia wynikania w odwrotnym kierunku przyjmujemy:

$$B_i = A (\Psi^i.\emptyset) \text{ dla } i \geq 0 \blacksquare$$

Z tej reguły wynika natychmiast reguła dla przypadku pojedynczej relacji zdefiniowanej równaniem stałopunktowym, czyli dla $n = 1$.

Reguła 3.5.2-2 Częściowa poprawność relacji zdefiniowanej stałopunktowo

Dla każdej funkcji ciągłej $\Psi : \text{Rel}(S) \mapsto \text{Rel}(S)$, jeżeli R jest najmniejszym rozwiązaniem równania $X = \Psi.X$, to dla każdych $A, B : S$ jest spełniona następująca reguła:

$$\begin{array}{l} \uparrow \text{istnieje rodzina warunków } \{B_i \mid i \geq 0\} \text{ taka, że} \\ (1) (\forall i \geq 0) [\text{ParPre } A] \Psi^i.\emptyset [\text{ParPost } B_i] \\ (2) U\{B_i \mid i \geq 0\} \subset B \\ \hline (3) [\text{ParPre } A] R [\text{ParPost } B] \end{array}$$

Można też tworzyć bardziej specyficzne reguły dla każdej z osobna konkretnej funkcji wielomianowej, na przykład dla zdefiniowanego w rozdziale 3.3 konstruktora rekurencji prostej. Poniżej dwie wersje takiej reguły.

Reguła 3.5.2-3 Częściowa poprawność relacji zdefiniowanej przez rekurencję prostą (wersja 1)

Dla każdych $H, T, E : \text{Rel}(S)$, jeżeli relacja R jest najmniejszym rozwiązaniem równania

$$X = HXT \mid E$$

to dla każdych $A, B \subset S$ jest spełniona następująca reguła:

$\uparrow$ istnieje rodzina warunków $\{B_i \mid i \geq 0\}$ taka że
 (1) $(\forall i \geq 0) A H^i E T^i \subset B_i$
 (2) $U\{B_i \mid i \geq 0\} \subset B$

 (3) $[ParPre A] R [ParPost B]$
 $\downarrow$

Dowód wynika bezpośrednio z reguły 3.5.2-2 oraz z faktu, że w tym przypadku, jak łatwo udowodnić:

$$R = U\{H^i E T^i \mid i \geq 0\} \quad \blacksquare$$

Przydatna może być też reguła jednokierunkowa o mocniejszej przesłance

Reguła 3.5.2-4 Częściowa poprawność relacji zdefiniowanej przez rekurencję prostą (wersja 2)

Dla każdych $H, T, E : Rel(S)$, jeżeli relacja R jest najmniejszym rozwiązaniem równania

$$X = HXT \mid E$$

to dla każdych $A, B \subset S$ jest spełniona następująca reguła:

$\uparrow$ (1) $(\forall Q) AQ \subset B$ implikuje $A HQT \subset B$
 (2) $AE \subset B$

 (3) $[ParPre A] R [ParPost B]$
 $\downarrow$

Dowód Z (1) i (2) możemy udowodnić przez indukcję, że dla każdego $i \geq 0$

$$A (H^i E T^i) \subset B$$

a stąd przez sumowanie stronami otrzymujemy (3). $\blacksquare$

Ta reguła może też być zapisana w sposób alternatywny (na co zwrócił mi uwagę Andrzej Tarlecki):

Reguła 3.5.2-4A Częściowa poprawność relacji zdefiniowanej przez rekurencję prostą (wersja 3)

Dla każdych $H, T, E : Rel(S)$, jeżeli relacja R jest najmniejszym rozwiązaniem równania

$$X = HXT \mid E$$

to dla każdych $A, B \subset S$ jest spełniona następująca reguła:

$\uparrow$ (1) $AH \subset A$ oraz $BT \subset B$
 (2) $AE \subset B$

 (3) $[ParPre A] R [ParPost B]$
 $\downarrow$

Dowód Z (1) i (2) wynikają zawierania $A (H^i E T^i) \subset AE T^i \subset B T^i \subset B$. Można też pokazać, że warunek (1A) implikuje warunek (1). $\blacksquare$

Z obu powyższych reguł, kładąc $H = [C]P$, $T = [S]$ oraz $E = [\neg C]$, można natychmiast wywieść odpowiednie reguły dla iteracji typu while-do-od.

Reguła 3.5.2-5 Częściowa poprawność pętli while-do-od (wersja 1)

Dla każdej relacji $P : \text{Rel}(S)$ oraz dla każdego rozłącznych $C, \neg C \subset S$, jeżeli relacja R jest najmniejszym rozwiązaniem równania

$$X = [C]PX \mid [\neg C],$$

to dla każdego $A, B \subset S$ jest spełniona następująca reguła:

$$\begin{array}{l} \uparrow \text{ istnieje rodzina warunków } \{B_i \mid i \geq 0\} \text{ taka że} \\ (1) (\forall i \geq 0) A ([C]P)^i [\neg C] \subset B_i \\ (2) \bigcup \{B_i \mid i \geq 0\} \subset B \\ \hline (3) [\text{ParPre } A] \ R \ [\text{ParPost } B] \\ \downarrow \end{array}$$

Reguła 3.5.2-6 Częściowa poprawność pętli while-do-od (wersja 2)

Dla każdej relacji $P : \text{Rel}(S)$ oraz dla każdego rozłącznych $C, \neg C \subset S$, jeżeli relacja R jest najmniejszym rozwiązaniem równania

$$X = [C]PX \mid [\neg C]$$

to dla każdego $A, B \subset S$ jest spełniona następująca reguła:

$$\begin{array}{l} \uparrow (1) (\forall Q) AQ \subset B \text{ implikuje } A [C]QR \subset B \\ (2) A[\neg C] \subset B \\ \hline (3) [\text{ParPre } A] \ R \ [\text{ParPost } B] \\ \downarrow \end{array}$$

W literaturze jest też powszechnie znana reguła jeszcze innej postaci, którą tym razem zapiszę posługując się notacją pre- i postwarunków:

Reguła 3.5.2-7 Częściowa poprawność pętli while-do-od (wersja 3)

Dla każdej relacji $P : \text{Rel}(S)$ oraz każdego $A, B, C, \neg C \subset S$, jeżeli $C \cap \neg C = \emptyset$, to jest spełniona następująca reguła:

$$\begin{array}{l} \uparrow \text{ istnieje } N \subset S \text{ (zwane niezmiennikiem pętli) takie, że:} \\ (1) [\text{ParPre } N \cap C] \ P \ [\text{ParPost } N] \\ (2) A \subset N \\ (3) N [\neg C] \subset B \\ \hline (4) [\text{ParPre } A] \ \text{while } (C, \neg C) \text{ do } P \text{ od } [\text{ParPost } B] \\ \downarrow \end{array}$$

Dowód. Niech będą spełnione (1) – (3). Z (1) przez indukcję

$$N([C]P)^i \subset N \text{ dla wszystkich } i \geq 0$$

Stąd i z (2)

$$A([C]P)^i \subset N \text{ dla wszystkich } i \geq 0$$

a więc z (3)

$$A([C]P)^i[\neg C] \subset N[\neg C] \subset B \text{ dla wszystkich } i \geq 0$$

Sumując to zawieranie stronami otrzymujemy (4). Przyjmijmy teraz, że spełnione jest (4) i oznaczmy

$$(5) N = A([C]P)^*$$

Stąd i z (4) otrzymujemy $N[\neg C] \subset B$, a stąd (3). Z kolei (5) jest równoważne

$$N = A \mid A([C]P)^+,$$

skąd (2). Dla udowodnienia (1) zauważmy, że:

$$(N \cap C)P = N[C]P = A[C]P \mid A([C]P)^+[C]P = A([C]P)^+ \subset N \quad \blacksquare$$

3.6 Poprawność całkowita

Reguły dla poprawności całkowitej pozwalają na dowodzenie, że jeżeli spełnione są odpowiednie warunki początkowe, to przynajmniej jedno wykonanie programu zakończy działanie ze spełnionymi warunkami końcowymi. Pamiętajmy, że w ogólnym przypadku nieokreśloność wyniku końcowego programu może oznaczać zarówno pojawienie się sygnału błędu, jak i nieskończone obliczenie. Pamiętajmy też, że w przypadku niedeterministycznego programu R całkowita poprawność

$$[\text{TotPre } A] R [\text{TotPost } B] \text{ czyli } A \subset RB$$

oznacza jedynie, że dla każdego $a : A$ istnieje wykonanie programu rozpoczynające się od a i kończące się w B , mogą jednakże istnieć obliczenia rozpoczynające się od a i bądź kończące się poza B , bądź też nie kończące się w ogóle. Jeżeli jednak R jest funkcją, to całkowita poprawność oznacza, że dla każdego $a : A$ rozpoczynające się od niego (jedyne) obliczenie kończy się w B .

3.6.1 Złożenie i rozgałęzienie

Reguła 3.6.1-1 Całkowita poprawność złożenia

Dla dowolnych $A, D \subset S$ oraz $P, Q : \text{Rel}(S)$ jest spełniona następująca reguła:

$$\begin{array}{l} \uparrow \text{istnieją warunki } B \text{ i } C \text{ takie, że} \\ (1) [\text{TotPre } A] \quad P \quad [\text{TotPost } B] \\ (2) [\text{TotPre } C] \quad Q \quad [\text{TotPost } D] \\ (3) B \subset C \\ \hline (3) [\text{TotPre } A] \quad PQ \quad [\text{TotPost } D] \end{array}$$

Dowód. Dwa pierwsze stojące nad kreską założenia wyrażone algebraicznie mają następującą postać:

$$(1) A \subset PB$$

$$(2) C \subset QD$$

Stąd natychmiast

$$A \subset PB \subset PC \subset P(QD) = (P;Q)D$$

Założmy teraz, że $A \subset (PQ)D$ co można zapisać jako $A \subset P(QD)$. Przyjmując $B = C = QD$ otrzymujemy (1) i (2). ■

Reguła 3.6.1-2 Całkowita poprawność if-then-else

Dla dowolnych $A, D, C, \neg C \subset S$ oraz $P, Q : \text{Rel}(S)$, jeżeli $C \cap \neg C = \emptyset$, to jest spełniona następująca reguła⁴³:

$$\begin{array}{l} \uparrow (1) [\text{TotPre } A \cap C] \quad P \quad [\text{TotPost } B] \\ (2) [\text{TotPre } A \cap \neg C] \quad Q \quad [\text{TotPost } B] \\ (3) A \subset C \mid \neg C \\ \hline \downarrow (4) [\text{TotPre } A] \quad \text{if } (C, \neg C) \text{ then } P \text{ else } Q \text{ fi} \quad [\text{TotPost } B] \end{array}$$

Dowód. Niech:

$$(1) A \cap C \subset PB$$

$$(2) A \cap \neg C \subset QB$$

$$(3) A \subset C \mid \neg C$$

Stąd:

$$[C] (A \cap C) \subset [C] PB$$

$$[\neg C] (A \cap \neg C) \subset [\neg C] QB$$

Dodając te zawierania stronami otrzymujemy:

$$[C] (A \cap C) \mid [\neg C] (A \cap \neg C) \subset [C] PB \mid [\neg C] QB = ([C]P \mid [\neg C]Q) B$$

Są też prawdziwe równości

$$[C] (A \cap C) = A \cap C$$

i analogicznie dla $\neg C$. Stąd i z (3)

$$[C] (A \cap C) \mid [\neg C] (A \cap \neg C) = (A \cap C) \mid (A \cap \neg C) = A$$

a więc ostatecznie

$$(4) A \subset [C] PB \mid [\neg C] QB$$

Z kolei z (4) wynika natychmiast $A \subset C \mid \neg C$, a z (4) i z rozłączności C i $\neg C$ wynikają (1) i (2). ■

Na koniec jeszcze trzy proste reguły związane z pre- i postwarunkami analogiczne do reguł dla częściowej poprawności:

⁴³ Zauważmy, że w przypadku predykatów dwuwartościowych, tj. określonych dla każdego stanu, warunek (3) nie jest potrzebny, gdyż $C \mid \neg C = S$.

Reguła 3.6.1-3 Wzmocnienie całkowitego prewarunku

Dla każdej $P : \text{Rel}(S)$ oraz każdego $A, B, C \subset S$ jest spełniona następująca reguła:

$$\frac{\begin{array}{l} [\text{TotPre } A] \quad P \quad [\text{TotPost } B] \\ C \subset A \end{array}}{[\text{TotPre } C] \quad P \quad [\text{TotPost } B]}$$

Reguła 3.6.1-4 Osłabienie całkowitego postwarunku

Dla każdej $P : \text{Rel}(S)$ oraz każdego $A, B, C \subset S$ jest spełniona następująca reguła:

$$\frac{\begin{array}{l} [\text{TotPre } A] \quad P \quad [\text{TotPost } B] \\ B \subset C \end{array}}{[\text{TotPre } A] \quad P \quad [\text{TotPost } C]}$$

Reguła 3.6.1-5 Koniunkcja warunków

Dla każdej $P : \text{Rel}(S)$ oraz każdego $A, B, C, D \subset S$ jest spełniona następująca reguła:

$$\frac{\begin{array}{l} [\text{TotPre } A] \quad P \quad [\text{TotPost } B] \\ [\text{TotPre } C] \quad P \quad [\text{TotPost } D] \end{array}}{[\text{TotPre } A \cap C] \quad P \quad [\text{TotPost } B \cap D]}$$

3.6.2 Rekurencja i iteracja

Podobnie jak w przypadku poprawności częściowej dyskusję tych reguł rozpocznę od przypadku najogólniejszego.

Reguła 3.6.2-1 Całkowita poprawność wektora relacji zdefiniowanego stałopunktowo

Dla każdej wektorowej funkcji ciągłej $\Psi : \text{Rel}(S)^{cn} \mapsto \text{Rel}(S)^{cn}$, jeżeli $\mathbf{R}$ jest najmniejszym rozwiązaniem równania $\mathbf{X} = \Psi.\mathbf{X}$, to jest spełniona następująca reguła, gdzie $\emptyset = (\emptyset, \dots, \emptyset)$ jest n -elementowym wektorem relacji pustych.

$$\frac{\begin{array}{l} \text{istnieje rodzina warunków } \{\mathbf{A}_i \mid i \geq 0\} \text{ taka, że} \\ (1) (\forall i \geq 0) [\text{TotPre } \mathbf{A}_i] \Psi^i.\emptyset [\text{TotPost } \mathbf{B}] \\ (2) \mathbf{A} \subset \mathbf{U}\{\mathbf{A}_i \mid i \geq 0\} \end{array}}{(3) [\text{TotPre } \mathbf{A}] \quad \mathbf{R} \quad [\text{TotPost } \mathbf{B}]}$$

Dowód. Jeżeli $\mathbf{R}$ jest najmniejszym punktem stałym Ψ , to z ciągłości Ψ wynika, że

$$(4) \mathbf{R} = \mathbf{U}\{\Psi^i.\emptyset \mid i \geq 0\}$$

Dodając stronami zawierania (1) otrzymujemy

$$\mathbf{U}\{\mathbf{A}_i \mid i \geq 0\} \subset \mathbf{U}\{\Psi^i.\emptyset \mid i \geq 0\} \mathbf{B}$$

a stąd i z (2) otrzymujemy (3). Załóżmy teraz, że $A \subset RB$ co oznacza, że

$$A \subset \bigcup \{\Psi^i.\emptyset \mid i \geq 0\} B$$

Niech dla $i \geq 0$

$$A_i = (\Psi^i.\emptyset) B$$

Wtedy oczywiście $A_i \subset (\Psi^i.\emptyset) B$. ■

Z tej reguły dla $n = 1$ otrzymujemy regułę:

Reguła 3.6.2-2 Całkowita poprawność relacji zdefiniowanej stałopunktowo

Dla każdej funkcji ciągłej $\Psi : \text{Rel}(S) \mapsto \text{Rel}(S)$, jeżeli R jest najmniejszym rozwiązaniem równania $X = \Psi.X$, to jest spełniona następująca reguła

$$\begin{array}{l} \uparrow \text{istnieje rodzina warunków } \{A_i \mid i \geq 0\} \text{ taka, że} \\ (1) (\forall i \geq 0) [\text{TotPre } A_i] \Psi^i.\emptyset [\text{TotPost } B] \\ (2) A \subset \bigcup \{A_i \mid i \geq 0\} \\ \hline (3) [\text{TotPre } A] R [\text{TotPost } B] \end{array}$$

Reguła 3.6.2-3 Całkowita poprawność procedury zdefiniowanej przez rekurencję prostą (wersja 1)

Jeżeli relacja R jest najmniejszym rozwiązaniem równania $X = H X T \mid E$ to jest spełniona następująca reguła:

$$\begin{array}{l} \uparrow \text{istnieje rodzina warunków } \{A_i \mid i \geq 0\} \text{ taka że} \\ (1) (\forall i > 0) A_i \subset H^i E T^i B \\ (2) A \subset \bigcup \{A_i \mid i \geq 0\} \\ \hline (3) [\text{TotPre } A] R [\text{TotPost } B] \end{array}$$

Dowód wynika bezpośrednio z reguły 3.6.1-2 oraz z faktu, że

$$R = \bigcup \{H^i E T^i \mid i \geq 0\} \quad \blacksquare$$

Reguła 3.6.2-4 Całkowita poprawność procedury zdefiniowanej przez rekurencję prostą (wersja 2)

Jeżeli relacja R jest najmniejszym rozwiązaniem równania $X = HXT \mid E$ to jest spełniona następująca reguła:

$$\begin{array}{l} (1) (\forall Q) (A \subset Q B \text{ implikuje } A \subset HQT B) \\ (2) A \subset EB \\ \hline (3) [\text{TotPre } A] R [\text{TotPost } B] \end{array}$$

Dowód. Z (1) i (2) możemy udowodnić przez indukcję, że dla każdego $i \geq 0$

$$A \subset (H^i E T^i) B$$

a stąd przez sumowanie stronami otrzymujemy (3). ■

Dla sformułowania reguły całkowitej poprawność pętli **while-do-od** wprowadzę kolejne pojęcie. Powiem, że komponenty pętli $(C, \neg C, P)$ spełniają *warunek stopu* w D , gdy

$$D \subset ([C]P)^*[\neg C]S$$

Zauważmy, że jeżeli P jest funkcją, to warunek stopu oznacza, że każde wykonanie instrukcji **while** $(C, \neg C)$ **do** P **od**, które rozpoczyna się w D , zakończy się w skończonym czasie. Rzeczywiście, jeżeli $s : D$, to

$$s : ([C]P)^*[\neg C]S$$

a wtedy istnieje takie $n \geq 0$, że $s : ([C]P)^n[\neg C]S$. Ponieważ P jest funkcją, a C i $\neg C$ są rozłączne, to n jest wyznaczone jednoznacznie, a więc jedyne wykonanie pętli, które rozpoczyna się w S odpowiada ciągowi n wykonań ciała pętli $[C]P$, po których następuje wykonanie $[\neg C]$.

Zauważmy, że jeżeli funkcja P reprezentuje program, który zawiera wewnętrzne pętle, to warunek stopu gwarantuje, że wykonanie również tych pętli zakończy się w skończonym czasie. Warunek stopu możemy też zapisać w postaci

$$[\text{TotPre } D] \text{ while } (C, \neg C) \text{ do } P \text{ od } [\text{TotPost } S]$$

Regułę całkowitej poprawności dla **while** ograniczę do programów deterministycznych, gdyż pozwala to na jej istotnie uproszczenie.

Reguła 3.6.2-5 Całkowita poprawność pętli while-do-od

Jeżeli $F : \text{Rel}(S)$ jest funkcją, to każdych $A, B, C, \neg C \subset S$, gdzie $C \cap \neg C = \emptyset$, jest spełniona następująca reguła:

$$\begin{array}{l} \uparrow \text{ istnieje warunek } N \text{ (niezmiennik) taki, że} \\ (1) [\text{TotPre } N \cap C] F [\text{TotPost } N] \\ (2) N \subset C \mid \neg C \\ (3) A \subset N \\ (4) N \cap \neg C \subset B \\ (5) (C, \neg C, F) \text{ spełniają warunek stopu w } N \\ \hline (6) [\text{TotPre } A] \text{ while } (C, \neg C) \text{ do } F \text{ od } [\text{TotPost } B] \end{array}$$

Dowód Załóżmy, że (1) – (5) są spełnione. Należy udowodnić, że

$$(6) A \subset ([C]F)^*[\neg C]B$$

Na podstawie (3) i (4) można to zadanie sprowadzić do dowodu, że

$$(7) N \subset ([C]F)^*[\neg C](N \cap \neg C)$$

Ponieważ jednak $[\neg C](N \cap \neg C) = [\neg C]N$, więc równoważną postacią (7) jest

$$(8) N \subset [\neg C]N \mid ([C]F)^*[\neg C]N$$

Niech więc $s : N$. Stąd i z (2) $s : N \cap C$ lub $s : N \cap \neg C$. Jeżeli $s : N \cap \neg C$, to $s : [\neg C]N$, co kończy dowód. Niech $s : N \cap C$. Na podstawie (5)

$$s : ([C]F)^+[\neg C]S.$$

czyli istnieje taki stan S_1 , że dla pewnego $n > 0$ zachodzi $S ([C]F)^n S_1$. Zauważmy teraz, że skoro F jest funkcją, to z warunku (1) odpowiadającego całkowitej poprawności wynika warunek odpowiadający częściowej poprawności

$$(N \cap C)F \subset N$$

co można zapisać w równoważnej postaci jako

$$N[C]F \subset N$$

Stąd $S_1 : N$, a zatem $S : ([C]F)^+[\neg C]N$, co kończy dowód wynikania z góry na dół. Załóżmy teraz, że spełnione jest (6) tzn.

$$A \subset ([C]F)^+[\neg C]B$$

Przyjmujemy:

$$N = ([C]F)^+[\neg C]B.$$

Wtedy:

- (3) wynika wprost z założenia (6),
- (2) wynika wprost z definicji N ,
- (5) wynika z definicji N oraz zawierania $B \subset S$,
- (4) wynika z rozłączności C i $\neg C$ oraz zależności

$$N \cap \neg C = ([C]F)^+[\neg C]B \cap \neg C = [\neg C]B \subset B$$

- (1) wynika z zależności

$$N \cap C = ([C]F)^+[\neg C]B \cap C = ([C]F)^+[\neg C]B = [C]F([C]F)^+[\neg C]B = [C]PN \subset PN$$

Co kończy dowód twierdzenia. ■

Zauważmy, że ponieważ F jest funkcją, to występujący w regule warunek N jest niezmiennikiem pętli w sensie częściowej poprawności.

W wielu sytuacjach praktycznych bywa technicznie niewygodnie dowodzić warunek stopu wprost z definicji. Wtedy pomocnym bywa lemat odwołujący się do pojęcia zbioru łańcuchowo ograniczonego.

Niech $(U, >)$ będzie zbiorem z określoną w nim relacją binarną. O tym zbiorze powiemy, że jest *łańcuchowo ograniczony*, jeżeli nie istnieje w nim nieskończony łańcuch elementów $u_1, u_2, \dots$ taki, że

$$u_i > u_{i+1} \text{ dla } i = 1, 2, \dots$$

Jak łatwo pokazać, jeżeli $(U, >)$ jest łańcuchowo ograniczony, to relacja $>$ jest:

- *przeciwwzrotna* tj. dla żadnego u nie zachodzi $u > u$, oraz
- *antysymetryczna*, tj. dla dowolnych u, w jeżeli $u > w$, to nie zachodzi $w > u$.

Lemat 3.6.2-1 *Jeżeli F jest funkcją, $N \cap C \subset FN$, $C \cap \neg C = \emptyset$ oraz $N \subset C \mid \neg C$, to składowe pętli $(C, \neg C, F)$ spełniają warunek stopu w N ,*

wtedy i tylko wtedy, gdy

istnieje taki zbiór łańcuchowo ograniczony $(U, >)$ oraz funkcja całkowita

$$K : S \mapsto U,$$

że $N \subset \text{dom}.K$ oraz dla każdych $a, b : S$

jeżeli $a F b$ to $K.a > K.b$

Dowód Przypuśćmy, że przy spełnionych założeniach lematu nie zachodzi

$$N \subset ([C]F)^*[\neg C]S.$$

Wtedy istnieje $s_0 : N$, tj. $s_0 : C \mid \neg C$, które nie należy do $([C]F)^*[\neg C]S$, a więc nie może należeć do $\neg C$, a więc $s_0 : N \cap C$. Zatem $s_1 = F.s_0$ jest określone i należy do N . W związku z tym $s_1 : C \mid \neg C$. Jednakże s_1 nie może należeć do $\neg C$, bo wtedy s_0 należałoby do

$$[C]F[\neg C]S$$

które jest podzbiorem $([C]F)^*[\neg C]S$. Rozumując w ten sposób można udowodnić istnienie nieskończonego ciągu stanów $(s_i : i = 0, 1, \dots)$ takiego, że

$$s_i F s_{i+1} \text{ dla } i = 0, 1, \dots$$

To jednak oznaczałoby istnienie nieskończonego ciągu

$$K.s_i > K.s_{i+1} \text{ dla } i = 0, 1, \dots$$

co jest niemożliwe.

Założmy teraz, że $N \cap C \subset FN$ i $N \subset C \mid \neg C$ oraz składowe pętli $(C, \neg C, F)$ spełniają warunek stopu w N , tj.

$$N \subset ([C]F)^*[\neg C]S.$$

Wtedy zbiór (N, F) jest łańcuchowo ograniczony, gdyż istnienie łańcucha nieskończonego

$$s_i F s_{i+1} \text{ dla } i = 0, 1, \dots$$

rozpoczynającego się w N oznaczałoby, że s_0 nie należy do $([C]F)^*[\neg C]S$. Zatem $U = N$, a K jest identycznością. ■

4 O modelach denotacyjnych ogólnie

Niniejszy rozdział stanowi wprowadzenie do ogólnej teorii denotacyjnych modeli języków programowania opartej na teorii algebr wielorodzajowych. W dalszej części książki będę pokazywał jak na gruncie tej teorii budować języki programowania zawierające dwie podstawowe kategorie narzędzi programistycznych:

1. *narzędzia aplikatywne* obejmujące wyrażenia danologiczne i typologiczne, których denotacje są funkcjami ze stanów pamięci w wartości i odpowiednio w typy,
2. *narzędzia imperatywne* obejmujące instrukcje i deklaracje, których denotacje są funkcjami ze stanów w stany.

4.1 Jak do tego doszło?

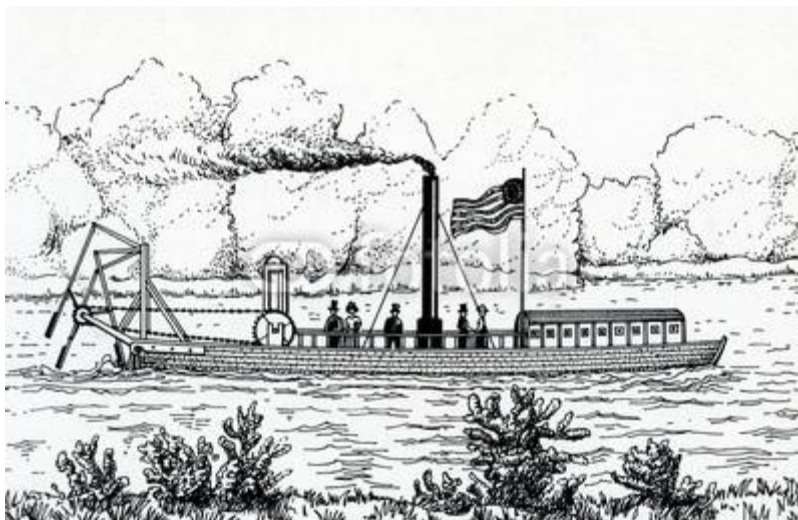
Idea, która przyświecała matematykom pracującym na stworzeniem formalnych opisów języków programowania, opierała się na założeniu, że każdy taki język można opisać na gruncie matematycznym za pomocą trzech obiektów:

1. *Syn* — *składnia* zwana też *syntaksą*, która w tej książce jest opisywana jako bezkontekstowa algebra składniowa,
2. *Den* — *denotacje* czyli znaczenia obiektów składniowych w tej książce opisywane jako algebra denotacji o tej samej sygnaturze co odpowiadająca jej algebra składni,
3. $\text{Sem} : \text{Syn} \mapsto \text{Den}$ — *semantyka* przypisująca denotacje elementom składniowym, która w tej książce jest traktowana jako wielorodzajowy homomorfizm pomiędzy wcześniej wspomnianymi algebrami.

Intuicyjnie rzecz ujmując, semantyka denotacyjna opisuje znaczenie każdego złożonego obiektu składniowego jako złożenie znaczeń jego składowych. Ta własność semantyki — zwana *kompozycyjnością* — pozwala opisywać złożone obiekty przez rozkładanie ich na części prostsze, a więc przez tzw. *indukcję strukturalną*.

W tym miejscu należy wyjaśnić, że denotacyjny model semantyki, który dla matematyków zawsze był wyborem dość oczywistym, nie był pierwszym modelem opisu języków programowania. Podobnie jak prototypem maszyny do szycia była mechaniczna ręka powtarzająca ruchy krawca, a na pierwszym parostatku maszyna parowa poruszała wiosłami, pierwszy formalny opis języka programowania był opisem wirtualnego komputera wykonującego programy tego języka⁴⁴.

⁴⁴ Metajęzyk służący do pisanie takich semantyk był rozwijany przez wiedeńskie laboratorium IBM i nosił nazwę Vienna Definition Language (VDL). Później część zespołu tego laboratorium stworzyła na Duńskim Uniwersytecie Technicznych (Danish Technical University) w Lyngby język do pisanie semantyk „bardziej denotacyjnych” pod nazwą Vienna Development Method (VDM) [10]. Ten język został wykorzystany m.in. do opisu semantyk języków programowania ADA[12] i CHILL [31]. W przypadku



Rys. 4.1-1 Parostatek poruszający wiosłami

Ten rodzaj semantyki, nazwany później *semantyką operacyjną*, zarzucono jednak po kilku latach eksperymentów, gdyż opis maszyny wirtualnej nie ustępował złożonością kompilatorowi języka, a na dodatek nie był opisem maszyny rzeczywistej⁴⁵.

Jednakże droga do semantyk denotacyjnych też nie była prosta. Jak już wspomniałem wcześniej, pierwsze teoretyczne modele denotacyjne języków programowania charakteryzowały się dużą złożonością matematyczną. Wynikała ona z przyjętego przez ówczesnych matematyków założenia, że immanentną cechą języków programowania „wyższego poziomu” — a więc poziomu wyższego od assemblera, nie mówiąc już o języku maszyny — jest obecność w języku dwóch mechanizmów:

1. instrukcji skoku **goto** pozwalającej przenieść wykonanie programu z dowolnej linii kodu do dowolnej innej; ta konstrukcja była obecna w praktycznie wszystkich językach programowania epoki lat 1960/70, a wywodziła się z języków niskiego poziomu, gdzie była jedynym narzędziem budowania logicznej struktury programu,
2. procedur, które mogą przyjmować jako parametry inne procedury, a w szczególności same siebie; ta konstrukcja była obecna w języku Algol 60, który przez akademickich informatyków był uznawany za niepodważalny standard dla języków programowania przyszłości.

Konieczność opisania **goto** doprowadziła do technicznie dość złożonego modelu tzw. *kontynuacji*⁴⁶, w którym skoki dawały się co prawda opisać, jednak kosztem bardzo dużej złożoności wszystkich mechanizmów języka. W tym modelu każda instrukcja była widziana nie jako operacja modyfikująca stan maszyny, ale jako operacja dopisująca swój efekt działania do tego, co stoi w programie za nią, czyli jej *kontynuacji*. Znaczenie programu budowano więc niejako od końca. Semantyka kontynuacyjna była nie tylko technicznie złożona, ale przede wszystkim

pierwszego z nich, który w ambicji jego twórców miał się stać uniwersalnym językiem programowania wszechczasów, proces pisanie semantyki języka pozwolił na wskazanie i usunięcie wielu nieścisłości w definicji języka, a sam opis semantyki — na napisanie pierwszego kompilatora języka Ada. Niestety, zarówno CHILL jak i Ada były nadmiernie złożone, co spowodowało, że dość szybko o nich zapomniano.

⁴⁵ Dla ścisłości należy dodać, że ta uwaga dotyczy jedynie języków programowania sekwencyjnego (nie obejmującego mechanizmów współbieżności), czyli takich, do których ograniczam się w niniejszej książce. W przypadku języków ze współbieżnością semantykę operacyjną stosował Gordon Plotkin ???

⁴⁶ Pierwszy wprowadził to pojęcie — zresztą pod nazwą „tail function” — Antoni Mazurkiewicz [55].

bardzo nieintuicyjna, co spowodowało, że nie spotkała się z większym zainteresowaniem ze strony budowniczych języków programowania. Niezależnie od tego już na przełomie 1960/1970 informatycy zaczęli sobie zdawać sprawę z niebezpieczeństwa jakie niesie ze sobą używanie instrukcji **goto** (por. [37]). Programy z dużą ilością „gotusów” były bardzo trudne do analizy, co powodowało, że często zachowywały się niezgodnie z intencją ich twórców. W rezultacie zaczęto promować tzw. *programowanie strukturalne* (rozdział 3.2), w którym **goto** eliminowano na rzecz takich konstruktorów programowych jak *if-then-else* oraz *while*.

Model kontynuacyjny, choć bardzo złożony, był oparty na tradycyjnej matematyce. Tego nie można już było powiedzieć o modelu służącym do opisu procedur, mogących przyjmować same siebie jako parametry aktualne. Należy w tym miejscu podkreślić, że nie chodziło tu o procedury rekurencyjne, które wywołują siebie w swoim ciele — takie konstrukcje dają się znakomicie opisywać równaniami stałopunktowymi — ale o konstrukcje typu $f(f)$, gdzie funkcja przyjmuje samą siebie jako swój argument. Taka konstrukcja była matematykom nieznana, gdyż nie daje się jej opisać na gruncie klasycznej teorii mnogości, nie mówiąc już o tym, że matematykom takie funkcje nigdy nie były potrzebne. Warto też zauważyć, że gdyby przez F oznaczyć zbiór samoaplikowalnych funkcji, to musiałby on spełniać stałopunktowe równanie dziedzinowe postaci

$$F = (F \mid A) \rightarrow B$$

w którym występuje nieciągły operator $\rightarrow$. Istnienie rozwiązania takiego równania nie jest więc gwarantowane przez twierdzenie Kleene’ego (rozdz.2.3).

W Algolu konstrukcja $f(f)$ była realizowana w ten sposób, że procedura f otrzymywała jako parametr nie literalnie „siebie”, ale kopię swojego kodu, która była wstawiana do treści procedury w procesie kompilacji. Była to tzw. *reguła kopiowania* (ang. *copy rule*). Matematyków lat 1960. początkowo zafascynowała ta możliwość, gdyż burzyła dotychczasowe pojęcie funkcji. W wyniku tej fascynacji powstała teoria *dziedzin zwrotnych* (ang. *reflexive domains*) stworzona przez Danę Scotta i Christophera Stracheya [64], a później szczegółowo opisana przez J.E. Stoya w monografii [63]. Pewna grupa matematyków rozwijała badania w tym kierunku, jednakże dla informatyków dziedziny zwrotne były jeszcze trudniejsze do zrozumienia i mniej intuicyjne niż kontynuacje. Dość szybko okazało się też, że możliwość przesyłania procedury do siebie samej jako parametru prowadzi do jeszcze większych niebezpieczeństw w praktyce programowania niż używanie **goto**. Zarzucono więc i tę konstrukcję, tak że w żadnym z używanych później języków programowania nie była już ona obecna. To spowodowało, że po jakimś czasie i matematycy przestali zajmować się dziedzinami zwrotnymi⁴⁷ oraz — niestety — również denotacyjnymi modelami języków programowania.

Opisywany w niniejszej książce model denotacyjny narodził się z dwóch nurtów badawczych. Pierwszy dotyczył odejścia od modelu kontynuacji i zastąpienia go modelem, w którym znaczeniami (denotacjami) instrukcji są funkcje przekształcające *wartościowania zmiennych*, czyli „stany komputera”, w nowe wartościowania.

Pojęcie wartościowania zmiennych, a więc funkcji przypisującej zmiennym ich wartości, było dobrze znane matematykom zajmującym się logiką matematyczną jeszcze od czasów

⁴⁷ Matematykom wyjaśniam, że idea dziedzin zwrotnych była w rzeczywistości odwzorowaniem reguły kopiowania. Autorzy tego modelu wykorzystali fakt, że funkcje definiowalne programami są obliczalne, można je więc „ponumerować” liczbami naturalnymi — każda funkcja f otrzymuje unikalny numer $n(f)$. Następnie utożsamiali oni funkcje z ich numerami, co oznaczało, że równość dziedzin zwrotnych nie była równością teoriomnogościową, ale szczególnym rodzajem izomorfizmu. W takim modelu konstrukcja $f(f)$ oznaczała w istocie $f(n(f))$ co na gruncie klasycznej teorii mnogości ma oczywiście sens. Powtórzono więc konstrukcję z Algolu 60, w którym za „numery” funkcji można uznać kody realizujących je programów.

pionierskich prac Alfreda Tarskiego [65]. W tamtych już czasach wyrażeniom matematycznym zwanym *termami*, przypisywano jako znaczenie funkcje, które każdemu wartościowaniu zmiennych

$$w : \text{Wartościowanie} = \{x, y, z, \dots\} \rightarrow \text{Wartość}$$

przypisywały wartość. Na przykład, znaczeniem wyrażenia arytmetycznego $2x+4y$ jest funkcja

$$F[2x+4y] : \text{Wartościowanie} \rightarrow \text{Liczba}$$

taka, że

$$F[2x+4y].w = 2 * w(x) + 4 * w(y).$$

Stąd już tylko krok do obserwacji, że znaczeniem instrukcji $x := 2x+4y$ może być funkcja, która zmienia wartościowanie w taki sposób, że wartością x w nowym wartościowaniu jest wartość wyrażenia $2x+4y$ w poprzednim. Tę konstrukcję zostałem m.in. w mojej pochodzącej z roku 1971 pracy [14], w której opisałem pewien prototyp semantyki denotacyjnej bardzo prostego języka programowania.

Inspiracją do porzucenia modelu dziedzin zwrotnych była dla mnie z kolei książka Michaela Gordona [45], w której autor potraktował dziedziny zwrotne Scotta-Stracheya jako „zwykłe zbiory” opatrując to takim oto komentarzem (str.29, wolne tłumaczenie moje)⁴⁸:

„Nie będziemy się w ogóle zajmowali matematyką wprowadzoną przez Scotta; nasze podejście do rekurencyjności jest podobne do podejścia inżynierów wobec równań różniczkowych, którzy zakładają, że takie równania mają rozwiązania, ale nie przejmują się matematycznym uzasadnieniem tej tezy.”

Tę książkę przeczytałem w roku 1981 w pociągu jadącym z Kopenhagi do Århus, gdzie zresztą jechałem na spotkanie z Peterem Mossesem — gorącym zwolennikiem teorii Scotta. Książka Gordona wywarła na mnie duże wrażenie, gdyż po raz pierwszy denotacyjny opis języka programowania czytałem ze zrozumieniem nie tylko jego warstwy matematycznej, ale też i informatycznej. Co prawda większa część książki była poświęcona semantyce kontynuacyjnej, jednakże już samo „potraktowanie” dziedzin zwrotnych jak „zwykłych zbiorów” stanowiło bardzo poważne uproszczenie. Odniosłem też wrażenie, że nie prowadziło do żadnych matematycznych problemów. Później zdałem sobie sprawę, że rzeczywiście tak było, gdyż Gordon właściwie nie zajmował się konstrukcją typu $f(f)$.

Podejście Michaela Gordona, choć intuicyjnie dość proste, nie miało jednak waloru matematycznej poprawności, gdyż założenie, że dziedziny zwrotne są „zwykłymi” zbiorami, jest po prostu nieprawdziwe. Nie było więc do końca wiadomo, czy budowany przez niego model semantyki języka nie będzie prowadził do fałszywych wniosków. Ten problem był szczególnie ważny w kontekście poszukiwań matematycznej metody dowodzenia poprawności programu, a więc jego zgodności z dokumentacją.

Aby uwolnić model Gordona od niebezpieczeństwa popadnięcia w sprzeczności, wspólnie z Andrzejem Tarleckim opublikowaliśmy w 1983 roku pracę [29], w której pokazaliśmy denotacyjny model języków programowania, w którym dziedziny są zbiorami w sensie klasycznej teorii mnogości, a instrukcje są transformacjami stanów maszyny, a nie funkcjami na kontynuacjach. To podejście zostało później rozwinięte w pracach nad językiem specyfikacji systemów

⁴⁸ We shall not discuss the mathematics involved in Scott's theory at all; our approach to recursive domains is similar to an engineering approach to differential equations, namely we assume they have solutions but don't bother with the mathematical justification.

oprogramowania **MetaSoft** [21], który był budowany w latach 1980. w Instytucie Podstaw Informatyki PAN. I na tym właśnie podejściu postanowiłem oprzeć moją książkę.

4.2 Od denotacji do składni

Wczesne prace dotyczące semantyki języków programowania były bez wyjątku poświęcone próbom budowaniu semantyk dla już istniejących języków. Przyjmowano więc milczące założenie, że najpierw budujemy język, czyli jego składnię, a dopiero później model matematyczny. Była w tym pewna logika, bo jak można budować model dla czegoś, czego jeszcze nie ma? Wszak astronomowie tworzyli mechanikę ciał niebieskich opisując istniejące słońca i planety.

Ten sposób myślenia ma jednak pewną lukę, gdyż informatyki — czym pisałem już wcześniej — nie można porównywać do astronomii, fizyki czy biologii, gdzie opisujemy otaczający nas świat. Budowanie języka programowania to tworzenie od początku pewnego „inżynierskiego bytu” takiego jak most, czy samolot. Czy jakimkolwiek inżynierowi przyszło by do głowy najpierw budować most „na oko i wyczucie”, by dopiero później liczyć wytrzymałość jego konstrukcji? Taki most z pewnością by się zawalił, o czym pisałem już w rozdziale 1.1.

W moim wykładzie postanowiłem odwrócić tradycyjną kolejność, zgodnie z którą najpierw budujemy składnię języka, a dopiero później jego model matematyczny, czyli denotację. Pokażę, jak budować język zaczynając od denotacji, by stąd w sposób systemowy generować składnię. Tak więc — najpierw treść, a dopiero później narzędzia jej wyrażania.

Wirtualny język programowania, który będę budował wraz z moimi czytelnikami, nazwałem **Lingua**, co w języku włoskim znaczy „język”. Wybrałem tę nazwę, aby upamiętnić okoliczności w jakich od października do grudnia 1969 roku pisałem pierwszą denotacyjną semantykę bardzo prostego języka programowania, później opublikowaną w *Dissertationes Mathematicae* [14] jako moją tezę habilitacyjną. Przez trzy miesiące jako stypendysta rządu włoskiego pracowałem w Istituto di Elaborazione dell’Informazione w Pizie. Nie znałem jeszcze wtedy ani prac Dany Scotta, ani pojęcia semantyki denotacyjnej, a mój model oparłem na teorii modeli będącej działem logiki matematycznej. Dopiero w osiemnaście lat później, bo w roku 1987 po raz pierwszy opisałem (w pracy [22]) konstrukcję od denotacji do składni.

4.3 Języki z serii Lingua

Jak już zapowiedziałem w rozdziale 4.2, metodę budowania denotacyjnego modelu języka programowania opiszę na przykładzie języka **Lingua**. Ten język będę budował warstwami, poczynając od mechanizmów aplikatywnych, by następnie wzbogacać je o kolejne konstrukcje. Każda kolejna warstwa będzie stanowiła wzbogacenie warstwy poprzedniej o nowe mechanizmy:

Lingua-A	aplikatywna część języków z serii Lingua obejmująca jedynie wyrażenia danologiczne i typologiczne, a więc jedynie modele dla danych i typów;
Lingua-1	instrukcje strukturalne, deklaracje zmiennych i definicje typów;
Lingua-2	procedury imperatywne z rekurencją wzajemną oraz procedury funkcyjne z rekurencją prostą;
LinguaV-2	narzędzia tworzenie poprawnych programów w Lingua-2 (V odpowiada „validated”);

- Lingua-3** programowanie obiektowe;
- Lingua-SQL** środowisko programistyczne pozwalające na przetwarzanie baz danych opartych na standardzie SQL.

Z perspektywy algebraicznej, algebra denotacji każdego z tych języków będzie rozszerzeniem algebry denotacji (w sensie określonym w rozdziale 2.11) języka poprzedzającego. Innymi słowy, każdy kolejny język będzie powstawał z poprzedniego przez rozszerzanie nośników algebry i/lub dodawanie nowych nośników algebry i/lub dodawanie nowych konstruktorów. Ta skalowalność algebr przekłada się w sposób naturalny na skalowalność ewentualnych implementacji.

W tym miejscu należy bardzo wyraźnie podkreślić, że opisywany w niniejszej książce język nie jest traktowany jako przyszły standard, a służy jedynie jako pole doświadczalne dla wskazywania metod, jak taki standard można by budować.

4.4 Po co nam denotacyjne modele języków programowania?

Denotacyjny model języka programowania powinien stanowić punkt wyjścia do realizacji trzech zadań:

1. zbudowania implementacji języka, a więc parsera i interpretera lub kompilatora,
2. stworzenia reguł budowania specyfikowanych programów poprawnych w tym języku,
3. napisania podręcznika użytkownika.

Budując taki model powinniśmy się stosować do pewnej ważnej, choć niesformalizowanej, zasady:

Zasada prostoty języka

Język programowania powinien być tak prosty i łatwy w użyciu, jak to tylko możliwe bez uszczerbku dla jego funkcjonalności oraz matematycznej jednoznaczności i pełności jego opisu. To samo dotyczy podręcznika języka, a także reguł budowania programów poprawnych.

Tę zasadę staram się realizować dbając o to, aby:

1. składnia języka była możliwie bliska językowi intuicyjnej matematyki, np. tam, gdzie jest to powszechnie przyjęte, dopuszczamy notację infiksową i zezwalamy na opuszczanie zbędnych nawiasów,
2. struktura języka (konstruktory programów) prowadziła do możliwie prostych reguł konstruowania programów poprawnych (rozdział 8),
3. semantyka języka była pisana przede wszystkim pod kątem przyszłego użytkownika, a nie pod kątem budowniczego implementacji; dla tego ostatniego należy zbudować odrębny model semantycznie równoważny z pierwszym; o co chodzi w tym przypadku postaram się wyjaśnić w dalszych częściach książki.

Na szczególną uwagę zasługuje punkt 2, gdyż prostota reguł budowania programów poprawnych bardzo wyraźnie przekłada się na lepsze rozumienie działania programu. Z tego faktu zaczęto zdawać sobie sprawę już w latach 1970., o czym świadczyły pojawiające się wtedy głosy za eliminacją instrukcji **goto**. Zrezygnowanie z niej stanowiło poważne uproszczenie struktur programowych, co przekładało się na zwiększenie niezawodności programów. Jednocześnie w niczym nie ograniczało funkcjonalności języków programowania.

Zgodnie z pkt.3 będę niekiedy — podobnie jak w potocznej matematyce — „zapominał” o różnicy pomiędzy składnią i denotacją. Będę więc mówić o wartości wyrażenia $x+y$, a nie o wartości denotacji tego wyrażenia, będę mówił, że instrukcja $x := y+1$ modyfikuje x , zamiast mówić, że denotacja tej instrukcji modyfikuje stan pamięci w punkcie x itp. Oczywiście na poziomie modelu nadal będę precyzyjnie odróżniał obiekty składniowe od denotacji.

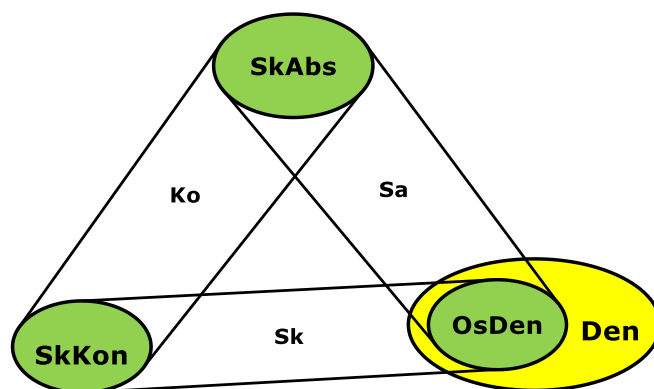
4.5 Pięć kroków do modelu denotacyjnego

Budując język **Lingua** będę się odwoływał do modelu algebraicznego, którego podstawy matematyczne opisałem w rozdziałach od 2.10 do 2.14. Ten model odpowiada diagramowi trzech algebr przedstawionemu na Rys. 4.5-1. Budujemy go w taki sposób, aby było spełnione równanie:

$$Sa = Ko \bullet Sk$$

Jeżeli tak właśnie jest, to mówimy, że nasz *model komutuje*.

Budowę denotacyjnego modelu języka rozpoczynamy od algebry denotacji **Den**. Jej konstruktory wyznaczają jednoznacznie część osiągalną algebry **OsDen**, która z kolei jednoznacznie wyznacza algebrę składni abstrakcyjnej **SkAbs**. W tych dwóch krokach pierwszy jest krokiem twórczym, w którym są podejmowane wszystkie najważniejsze decyzje dotyczące przyszłego języka, natomiast drugi jest w pełni algorytmizowalny⁴⁹.



Rys. 4.5-1 Algebraiczny model języka programowania

Jak widzieliśmy w rozdziale 2.12, składnia abstrakcyjna jest niezbyt wygodna dla przyszłego użytkownika języka. Aby uczynić ją bardziej przyjazną dla programisty buduje się *składnię konkretną* **SkKon**⁵⁰. W typowych sytuacjach polega to na zastąpieniu notacji prefiksowej

⁴⁹ Algorytm, który buduje składnię abstrakcyjną z denotacji, nie bierze oczywiście na wejściu abstrakcyjnego obiektu jakim jest algebra, ale jej opis w metajęzyku MetaSoft. Ta idea jest wyjaśniona na szczegółowym przykładzie w rozdziale 5.4.1.

⁵⁰ Ta nazwa została przyjęta w latach 1970. przez autorów pierwszych algebraicznych modeli języków programowania.

infiksową i opuszczeniu niektórych „zbędnych” nawiasów. Bardzo prosty przykład takiej transformacji pokazałem w rozdziale 2.13, gdzie składnia konkretna stanowiła algebrę podobną do składni abstrakcyjnej, a prowadzący do niej homomorfizm KO sklejał nie więcej niż Sa , istniał więc jednoznacznie wyznaczony homomorfizm

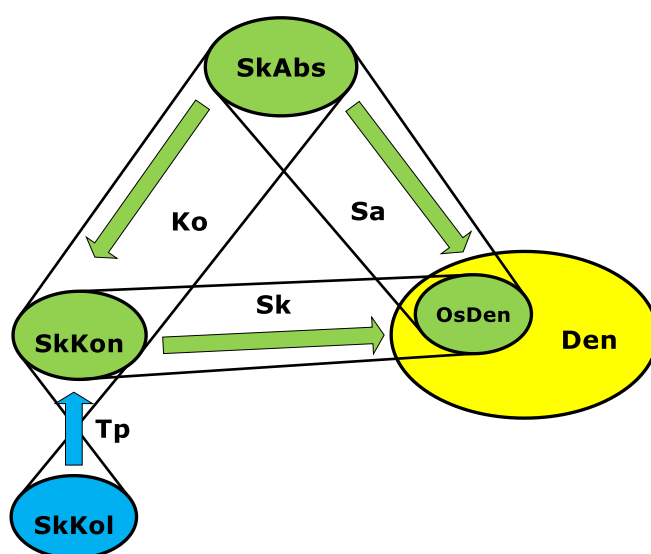
$$Sk : SkKon \mapsto OsDen$$

stanowiący semantykę denotacyjną składni konkretnej. W ten właśnie sposób powstaje główna część denotacyjnego modelu języka. Zauważmy, że krok od składni abstrakcyjnej do konkretnej jest krokiem twórczym — choć dość prostym — tu więc również podejmowane są istotne decyzje dotyczące przyszłego języka.

Przykład przejścia od składni abstrakcyjnej do konkretnej pokazany w rozdziale 2.13 polegał na opuszczeniu nawiasów w wyrażeniach arytmetycznych typu $((a + b) + c)$. Ta konstrukcja była jednak możliwa tylko dlatego, że w wyrażeniach dopuściłem jedynie działanie dodawania, które jak wiemy jest łączne⁵¹, tj.

$$(a + b) + c = a + (b + c) .$$

W rozdziale 2.14 pokazałem jednak przykład języka wyrażeń z dwiema operacjami — dodawania i mnożenia — co spowodowało, że dla zbudowania w pełni denotacyjnego modelu języka bez „zbędnych nawiasów”, trzeba było wprowadzić odpowiednie konstrukcje już na poziomie denotacji. Na dodatek, ten model był dość mało intuicyjny.



Rys. 4.5-2 Algebraiczny model języka ze składnią kolokwialną

Takie rozwiązania zastosowano w gramatykach języków Algol [61] i Pascal [48], których semantyki opisywano jednak całkiem nieformalnie w języku potocznym. Wychodzono wtedy z założenia, że gramatyka języka powinna służyć zarówno programiście jak i twórcy implementacji. Przyjęcie tej tezy oznacza jednak, że już budując algebrę denotacji powinniśmy myśleć o przyszłej składni konkretnej. W moim przekonaniu zaburza to naszą filozofię „od denotacji do składni”, gdzie powinniśmy przestrzegać zasady, że najpierw myślimy o treściach, a dopiero później o tym, jak wyrazić je na poziomie składni.

⁵¹ Jak zobaczymy w rozdziale

Alternatywą dla wyżej opisanego postępowania, którą krótko omówiłem w rozdziale 2.14, jest dopuszczenie w składni konkretnej konwencji notacyjnych, które nazywam *kolokwializmami* (Rys. 4.5-2). Wprowadzenie kolokwializmów do składni konkretnej SkKon prowadzi do *składni kolokwialnej* SkKol najczęściej niehomomorficznej z konkretną, a nawet o innej sygnaturze. Musi jednak istnieć algorytmizowalna transformacja

$$Tp : \text{SkKol} \mapsto \text{SkKon}$$

usuwająca kolokwializmy, np. dopisująca brakujące nawiasy. Tę transformację nazywam *transformacją przywracającą*. Oczywiście transformacja Tp nie musi być homomorfizmem w sensie zdefiniowanym w rozdziale 2.11.

W podręczniku programisty dla języka opartego na takim modelu składnia jest opisana gramatyką bezkontekstową składni konkretnej, którą uzupełniamy o nieformalnie opisane kolokwializmy. W przypadku tej drugiej wyjaśniamy na przykład, że pisząc wyrażenia arytmetyczne możemy opuszczać nawiasy zgodnie z przyjętym priorytetem mnożenia i dzielenia nad dodawaniem i odejmowaniem.

Przy takim postępowaniu budowniczy implementacji otrzymuje standardowy model denotacyjny języka wraz z formalną definicją transformacji Tp. Wykonywanie programu polega na wstępnej obróbce programu transformacją przywracającą, by następnie poddać go parsingowi i interpretacji.

Budowa tak rozumianego modelu algebraicznego języka dokonuje się w pięciu krokach, które określe wspólnym mianem *metody pięciu kroków*.

1. W pierwszym kroku budujemy algebrę denotacji Den obejmującą obiekty przyszłego języka oraz ich konstruktory. Przy definiowaniu tej algebry są podejmowane wszystkie najważniejsze decyzje dotyczące mechanizmów, jakie chcemy zawrzeć w języku. Projektant języka musi określić taki repertuar konstruktorów w Den (funkcji pomiędzy nośnikami), aby jej podalgebra osiągalna OsDen zawierała wszystkie obiekty, które chcemy w przyszłości opisywać strukturami składniowymi, tj. które chcemy oddać do dyspozycji programisty.
2. Sygnatura algebry Den wyznacza jednoznacznie algebrę składni abstrakcyjnej SkAbs i odpowiadający jej homomorfizm Sa, który będę nazywać *semantyką abstrakcyjną*. Krok od OsDen do SkAbs można więc w pełni zalgorytmizować. Ze strony projektanta ten etap nie wymaga żadnej kreatywności i może być wspomagany odpowiednim systemem komputerowym.
3. Składnia abstrakcyjna jest zwykle mało „przyjazna” dla przyszłego użytkownika języka, wymaga bowiem stosowania wielu nawiasów i jest ograniczona do notacji prefiksowej. Dla usunięcia tych niedogodności tworzy się tzw. składnię konkretną SkKon, tj. bliską tej, którą będą się posługiwali przyszli użytkownicy języka. Algebraicznie ta składnia jest homomorficznym obrazem składni abstrakcyjnej i musi być tak zbudowana, aby odpowiadający jej homomorfizm Ko — który nazywam *konkretyzacją* — sklejał nie więcej niż Sa. Tworzenie składni konkretnej jest drugim kreatywnym zadaniem projektanta języka i zwykle jest serią kroków, na którą składają się kolejne konkretyzacje składni abstrakcyjnej.
4. Jeżeli Ko skleja nie więcej niż Sa, to na mocy Tw.2.13-1 istnieje jednoznacznie wyznaczony (a więc jedyny) homomorfizm Sk ze składni konkretnej w algebrę denotacji, a dokładniej — na jej podalgebrę osiągalną. Ten homomorfizm będę nazywał *semantyką konkretną*, albo po prostu *semantyką* naszego języka. Jego opis daje się algorytmicznie wygenerować z opisów Sa i Ko.

5. W ostatnim kroku wprowadzamy do języka kolokwializmy i opisujemy transformację przywracającą Tp. Ten krok ma również charakter kreatywny.

Jak widzimy, kreatywne działania projektanta języka mają miejsce w krokach pierwszym, trzecim i piątym. Kroki drugi i czwarty mogą być zrealizowane algorytmicznie.

Po zbudowaniu modelu denotacyjnego języka programowania można przystąpić do budowania reguł konstruowania programów poprawnych, a więc do budowania — jak to kiedyś nazywano — Hoare’owskiej logiki programów. Temu tematowi będzie poświęcony rozdział 8.

4.6 Metakonwencje notacyjne i językowe

W opisie przykładowego języka **Lingua** będziemy używali trzech językowych poziomów formalizacji, które dla odróżnienia jednego od drugiego będą zapisywane różnymi krojami pisma:

1. na poziomie składni konkretnej i kolokwialnej języka **Lingua** będzie używany Courier New,
2. na poziomie formalnych definicji pozostałych składowych naszego modelu, a więc algebr denotacji i składni oraz łączącej je semantyki, będzie używany Arial, natomiast konwencje notacyjne będą pochodzić z na wpół sformalizowanego języka **MetaSoft**, o którym była już mowa w rozdziale 4.1⁵²,
3. na poziomie komentarzy i opisów nieformalnych będzie używany Times New Roman.

Indeksy, które w tradycyjnej matematyce pisze się obniżone i zmniejszoną czcionką, np. a_i , będę traktował jako argumenty funkcji pisząc $a.i$, gdzie a jest traktowane jako funkcja, natomiast i — jako jej argument.

Ze względu na wielką mnogość symboli występujących w sformalizowanych opisach języków programowania, w miejsce typowych dla matematyki abstrakcyjnej symboli jednoliterowych $a, b, c, \dots$ używam — podobnie jak w językach programowania — oznaczeń wieloliterowych typu *ide*, *sta*, *skł*, *śro* itp.

Nazwy zbiorów piszę zawsze z dużej litery, np. Rzeczywista, a nazwy ich elementów — z małej. Elementy zbiorów oznaczam (najczęściej trzyliterowymi) nazwami nawiązującymi do nazwy zbioru, np. *sta* : Stan.

Idąc śladem konwencji stosowanej w Vienna Development Method [10], metazmienne przebiegające dziedziny „zapowiadam” w definicjach dziedzin. Piszę więc np.

ide : Identyfikator = Litera © Znak*

wrt : Wartościowanie = Identyfikator $\Rightarrow$ Dana

na wskazanie, że *ide* jest metazmienną przebiegającą dziedzinę Identyfikator, a *wrt* metazmienną przebiegającą dziedzinę Wartościowanie. Aby ułatwić posługiwanie się tymi oznaczeniami, na końcu książki umieszczam ich spis.

Jak już wyjaśniałem w rozdziale 2.8, wartości będące napisami ujmuję w apostrofy dla odróżnienia ich od metazmiennych. Tak więc *ide* jest metazmienną przyjmującą jako swoje wartości identyfikatory, a ‘*abcd*’ — konkretnym napisem, który może być wartością metazmiennej *ide*.

⁵² O tym języku piszę, że jest „na wpół sformalizowany”, gdyż nie został stworzony dla niego model denotacyjny.

Dla skrócenia bardzo licznych w mojej książce definicji warunkowych przyjmuję konwencję, że zamiast pisać

warunek-1 ➔ wynik-1

...

warunek-n ➔ wynik-n

będę pisać

warunek-i ➔ wynik-i dla $i = 1;n$.

5 Algebraiczno-denotacyjny model struktur danych

5.1 Ogólna idea modelu

We wczesnych językach programowania takich jak Fortran, Algol, Pascal czy Cobol pojęcie typu danej wprowadzono w pierwszym rzędzie po to, aby przypisując zmiennej typ alokować dla niej odpowiedni obszar pamięci. Ze zmienną typu boolowskiego wiązano prosty jednobitowy rejestr, ze zmienną liczbową — rejestr wielobitowy, a wreszcie ze zmienną typu tablica — obszar pamięci uzależniony od rozmiaru tej tablicy. Z biegiem czasu okazało się jednak, że przypisywanie zmiennym typów pozwala nie tylko na oszczędne gospodarowanie pamięcią, ale przyczynia się też do lepszego panowania przez programistę nad funkcjonalnością programu.

Dziś, gdy gospodarowanie pamięcią nie ma już tak krytycznego znaczenia (poza np. bazami danych), ten drugi aspekt pozostaje nadal ważny. Zdarza się też, że typ danej zmienia się wraz z tą daną; np. dodanie nowego atrybutu do rekordu rozbudowuje nie tylko zawartość rekordu, ale także jego typ. W takich sytuacjach potrzebny jest mechanizm śledzenia na bieżąco typu danej.

W denotacyjnych modelach dla języków z grupy **Lingua** ten mechanizm zostanie zrealizowane przez zbudowanie dwóch algebr:

- algebry kompozytów, której elementami będą pary składające się z danej i jej korpusu (struktury)⁵³,
- algebry typów, której elementami będą pary składającej się z korpusu i predykatu zwanego *jarzmem*.

Intuicyjnie typy kojarzymy za zbiorami danych. Typ liczbowy to zbiór liczb, a listowy — zbiór list. Jednakże operowanie na tak rozumianych typach nie byłoby zbyt wygodne. W budowanym niżej modelu typy będą więc pewnymi niezależnymi bytami matematycznymi, z których każdy będzie jedynie wyznaczał pewien zbiór danych zwany jego *klanem*. Ten model pozwoli na wprowadzenie do języka narzędzi służących programiście nie tylko do definiowania własnych typów o zróżnicowanych cechach, ale też do zapisywania ich w pamięci komputera do dalszego wykorzystania. W szczególności pozwoli to też na budowanie złożonych typów metodą wstępującą, a więc na tworzenie typów złożonych z typów prostych.

W toku badań nad tak rozumianymi typami — i prawdę mówiąc po wielu nieudanych próbach, co zajęło mi blisko rok pracy — doszedłem do przekonania, że traktując typy jako opisy zbiorów danych wygodnie jest wyróżnić w nich dwie niezależne części:

⁵³ Pewną inspiracją dla wprowadzenia tej konstrukcji była dla mnie idea wykorzystana w definicji języka programowania Ada napisanej w metajęzyku VDM (patrz [10] i [12]). W tamtym jednak przypadku wprowadzono dwie semantyki, tzw. semantykę dynamiczną odpowiedzialną za wyliczanie danych oraz semantykę statyczną odpowiedzialną za wyliczanie typów. Pierwsza opisywała wykonanie programu, a druga — jedną z procedur realizowanych w czasie kompilacji. Takie rozwiązanie może być wygodne dla budowniczego implementacji, jest jednak dość odległe od perspektywy programisty, której staram się trzymać w mojej książce.

1. opis struktury danej, np. liczba, słowo, tablica, lista, rekord czy drzewo, który nazwę *korpusem* danej,
2. opis innych własności danej, np. liczba z określonego przedziału, tablica liczb całkowitych, rekord, w którym suma liczb przypisanych wskazanym atrybutom nie przekracza określonej wielkości; opisy tego typu własności będą specyficznymi predykatami, które nazwę *jarzmami*.

Pierwszy sposób na opisanie typu danej znany jest z większości języków programowania, drugi jest charakterystyczny dla języków bazodanowych, np. opartych na standardzie SQL. Zresztą w tym drugim przypadku wykorzystywane są zwykle oba sposoby.

Wprowadzenie opisanych wyżej pojęć do modelu denotacyjnego wymaga zdefiniowania pięciu algebr składających się na *algebraiczny model struktur danych*:

1. *algebra danych*, a więc algebra, której elementami są liczby, dane boolowskie, słowa, listy, rekordy itp.
2. *algebra korpusów*, których elementy opisują wewnętrzną strukturę danej; np. liczby mają strukturę inną niż tablice liczb,
3. *algebra kompozytów* będącymi parami (dana, korpus), w których korpus opisuje strukturę danej dana,
4. *algebra transferów* będących jednoargumentowymi funkcjami z kompozytów w kompozyty; wspomniane wcześniej *jarzma* to transfery przyjmujące kompozyty boolowskie jako wartości,
5. *algebra typów* będących parami (korpus, jarzmo); w tym przypadku nie zakładam związku pomiędzy korpusem i jarzmem.

Obok wskazywanych przez powyższe algebry pięciu rodzajów obiektów tworzą jeszcze szósty, który nazywam *wartością*, a jest nim para (dana, typ) gdzie dana jest typu typ. Na wartości można też spojrzeć jako na trójki (dana, korpus, jarzmo), gdzie:

- dana odpowiada korpusowi co do struktury,
- kompozyt (dana, korpus) spełnia jarzmo.

Dla wartości nie tworzę odrębnej algebry, gdyż jej operacje zostaną implícite opisane w algebrze denotacji wyrażeń. To zabieg o charakterze czysto technicznym mający na celu uproszczenie opisu naszego modelu.

Nad tak zbudowanym modelem struktur danych buduję teraz algebry denotacji *wyrażeń danologicznych* i *typologicznych*, których elementami są funkcje odwzorowujące stany odpowiednio w kompozyty i w typy:

$\text{dwd} : \text{DenWyrDan} = \text{Stan} \rightarrow \text{Kompozyt} \mid \text{Błąd}$

$\text{dwt} : \text{DenWyrTyp} = \text{Stan} \mapsto \text{Typ} \mid \text{Błąd}$

Przyjęcie, że denotacje wyrażeń danologicznych są funkcjami częściowymi, wiąże się z faktem wprowadzenia na dalszych etapach budowania modelu procedur funkcyjnych (rozdział 7.5), których wywołania są wyrażeniami i które mogą generować obliczenia nieskończone.

Przyjmuję, że stany wiążą z identyfikatorami wartości i typy. W pierwszym przypadku mówię o *zmiennej danologicznej*, a w drugim — o *stałej typologicznej*. W praktyce oznacza to, że wartości przypisywane w programie zmiennym danologicznym mogą się zmieniać, natomiast typy przypisane stałym typologicznym pozostają niezmiennie przez cały czas wykonania programu.

Wybiegając nieco w przyszłość należy wyjaśnić, że choć zmienne danologiczne przechowują wartości, to efektem wykonania wyrażeń danologicznych będą kompozyty, a nie wartość. Jeżeli więc identyfikatorowi *ide* przypisano w stanie pamięci wartość (*dana*, *korpus*, *jarzmo*) to wykonanie instrukcji przypisania postaci

```
ide := wyrażenie
```

będzie oznaczało utworzenie nowego kompozytu (*dana-1*, *korpus-1*), a następnie przypisanie zmiennej *ide* wartości (*dana-1*, *korpus-1*, *jarzmo*), o ile tylko nowy kompozyt spełnia zastane *jarzmo*. Jeżeli *jarzmo* nie będzie spełnione, to zostanie wygenerowany sygnał błędu. Jak więc widzimy, *jarzma* nie będą brały udziału w ewaluacji wyrażeń, a posłużą jedynie do zachowania dyscypliny typologicznej na poziomie instrukcji.

W tym miejscu czytelnik może zadać pytanie, dlaczego nie postąpiłem podobnie z *korpusami*, które jednak włączyłem do mechanizmu wykonywania wyrażeń? Otóż postąpiłem tak dlatego, by do denotacji wyrażeń wprowadzić łatwo implementowalny mechanizm, który pozwoli na stwierdzenie, czy występujące w wyrażeniach argumenty operacji na danych „otrzymały” właściwe dane oraz na wyemitowanie sygnału błędu, jeżeli tak się nie stało. Na przykład, można w ten sposób stwierdzić, czy dane, które chcemy do siebie dodać są liczbami, czy może słowami lub rekordem i listą (szczegóły w rozdziale 5.2.3).

W kolejnych rozdziałach przedstawię pewną ograniczoną wersję mojego modelu, co jednak powinno pozwolić na wskazanie, jak można go rozbudowywać dla objęcia większości konstrukcji występujących w istniejących językach programowania lub też stworzenia rozwiązań nowych.

Na koniec należy podkreślić, że **Lingua-A**, który będziemy tu budować, nie stanowi przykładu samodzielnego języka programowania aplikatywnego, a jedynie przykład części aplikatywnej języka imperatywnego. W **Lingua-A** wyrażenia będą pobierać dane zapisane w stanach, ale nie będzie w nim mechanizmów pozwalających na modyfikowania stanów. Takie mechanizmy zostaną wprowadzone dopiero w **Lingua-1** (rozdział 6).

5.2 Algebry struktur danych

5.2.1 Algebra danych

Rozpocznę od zdefiniowania pewnej standardowej rodziny dziedzin danych, która stanie się fundamentem dla zbudowania przyszłej algebry denotacji wyrażeń. Należy podkreślić, że zdefiniowane niżej dziedziny są nadzbiorami przyszłych zbiorów danych generowalnych przez programy **Lingua**, co pozwala zdefiniować te dziedziny za pomocą prostych równań dziedzinowych (rozdział 2.7). A oto lista naszych dziedzin wyjściowych:

```
boo : Boolowska = {pp, ff}
lic  : Liczba    — zbiór wszystkich liczb o skończonych rozwinięciach dziesiętnych
sło  : Słowo     = {'}Alfabet*{'}
```

$$\text{lis} : \text{Lista} = \text{Dana}^{\text{c}*}$$

```
tab  : Tablica   = Liczba  $\Rightarrow$  Dana
rek  : Rekord    = Identyfikator  $\Rightarrow$  Dana
dan  : Dana      = Boolowska | Liczba | Słowo | Lista | Tablica | Rekord
ide  : Identyfikator — ustalony skończony podzbiór dziedziny Alfabetc+
```

Dziedzina Alfabet jest jakimś ustalonym skończonym zbiorem znaków niezawierającym cudzysłówów prostych, natomiast Identyfikator jest pewnym skończonym zbiorem słów nad alfabetem Alfabet. Słowo jest ciągiem (być może pustym) elementów zbioru Alfabet ujętym w cudzysłowy proste. Założę, że zbiory Alfabet oraz Identyfikator stanowią parametry naszego modelu i zostały raz na zawsze ustalone.

Zauważmy, że identyfikatory nie zostały zaliczone do danych. Identyfikatory występujące w rekordach będę nazywał *atrybutami rekordów*.

Pierwsze trzy dziedziny z naszej listy nazywam *dziedzinami prostymi*, a ich elementy *danymi prostymi*. Wprowadzam więc oznaczenie na tę dziedzinę:

dan : DanaProsta = Boolowska | Liczba | Słowo

Listy, tablice i rekordy tworzą *dziedziny strukturalne* zawierające *dane strukturalne*. Te ostatnie dzielą się znów na dwie klasy: krotki, którymi są listy oraz mapingi, którymi są tablice i rekordy.

Wszystkie zdefiniowane wyżej dziedziny poza dwiema ostatnimi będę nazywał *rodzajami danych*. Będę więc mówił, że dana jest *rodzaju liczbowego, słowowego, listowego* itp.

Zauważmy, że jak na razie lista może zawierać dane różniących się między sobą rodzajów, dziedziną tablicy może być dowolny skończony zbiór liczb niekoniecznie całkowitych, a rekord może wiązać z atrybutami dowolne dane. Wszystkie dane mogą też być „dowolnie duże”. W przyszłości wiele z tych cech danych zostanie ograniczonych na gruncie algebraicznym przez odpowiednie określenie konstruktorów algebr. Oczywiście konstruktory nie ograniczą nośników, ale wyznaczą ich podzbiory osiągalne (rozdział 2.12), a więc ten obszar elementów algebry, które będą generowane w trakcie wykonywania programów.

Zarówno listy jak też tablice i rekordy mogą być puste. To oczywiście założenie o charakterze czysto technicznym.

Zauważmy też, że tablice są jednowymiarowe, ponieważ jednak ich elementami mogą być również tablice, w praktyce możemy tworzyć tablice dowolnego wymiaru. Np. tablica dwuwymiarowa, to jednowymiarowa tablica tablic jednowymiarowych.

Po określeniu dziedzin danych należy zdefiniować operacje na danych, których wybór jest kluczową decyzją inżynierską podejmowaną w pierwszej fazie budowania języka. Poniżej określę więc listę takich operacji dla **Lingua**. Nie będzie ona zbyt bogata, by szczegóły techniczne (których i tak będzie dość dużo), nie przesłoniły idei budowanego modelu. Tę listę należy więc traktować jako przykładową lub jako parametr modelu.

Określone niżej operacje będę nazywał dalej *operacjami teoretycznymi*, gdyż nie dają się one „w pełni” zaimplementować. Na przykład teoretyczna operacja dzielenia liczb:

dziel : Liczba x Liczba $\rightarrow$ Liczba

może wygenerować dowolnie dużą liczbę lub też liczbę o dowolnie długim rozwinięciu dziesiętnym, a więc w obu przypadkach niereprezentowalną w komputerze. Tymczasem w większości języków programowania — i tak będzie też w **Lingua-A** — wartość wyrażenia:

x / y

nie może przekroczyć pewnej ustalonej wielkości, a ponadto powinna być „wyliczalna” w każdej sytuacji, a więc nawet wtedy, gdy wartości x i y nie są liczbami, a także gdy wartością y jest zero. Oczywiście w takich sytuacjach wyrażenie wygeneruje komunikat błędu. Te wszystkie ograniczenia zostaną wpisane do modelu na kolejnych etapach jego tworzenia.

Ustalanie listy operacji teoretycznych rozpoczynam od konstruktorów zeroargumentowych, których ideę i potrzebę wyjaśniłem w rozdziale 2.10.

twórz-id.ide : $\mapsto$ Identyfikator dla ide : Identyfikator

twórz-bo.bo : $\mapsto$ Boolowska dla bo : Boolowska

twórz-lic.lic : $\mapsto$ Liczba dla lic : LiczbaS

twórz-sł.sł : $\mapsto$ Słowo dla sł : SłowoS

W tym przypadku LiczbaS i SłowoS oznaczają podzbiory odpowiednio zbiorów liczb i słów, o których zakładam, że będą reprezentowalne składniowo, a więc, że będzie je można „wpisywać ręcznie” do programów (S od „składnia”). Czym są te zbiory, będzie zależało od implementacji. Na poziomie ogólnym zakładam jedynie, że są skończone. Nie zakładam jednak, ale też i nie wykluczam, by pozostałe konstruktory liczb i słów musiały generować dane jedynie z tych zbiorów. Dopuszczam natomiast taką sytuację, że nie wszystkie liczby i słowa, które mogą powstawać w wyniku wykonywania programów, mogą być również wpisywane do programów „z klawiatury”, tj. że mogą występować na poziomie składni.

Dziedzina identyfikatorów zawiera jedynie takie elementy, które można wpisywać z klawiatury, tu więc nie dodaję przyrostka „S”.

Pozostałe konstruktory mają niżej opisane arności i rodzaje. Jak już zapowiadałem, ograniczę się do ich mocno ułomnej listy. Dopuszczę też operacje częściowe, co wykracza poza pojęcie algebry opisane w rozdziale 2.11. To ostatnie założenie nie rodzi jednak żadnych problemów, gdyż algebra danych ma charakter pomocniczy, a jej konstruktory posłużą jedynie do zdefiniowania konstruktorów kompozytów, które będą już funkcjami całkowitymi.

oraz	: Boolowska x Boolowska	$\mapsto$ Boolowska	
lub	: Boolowska x Boolowska	$\mapsto$ Boolowska	
nie	: Boolowska	$\mapsto$ Boolowska	
równe	: DanaProsta x DanaProsta	$\mapsto$ Boolowska	
mniejsze	: Liczba x Liczba	$\mapsto$ Boolowska	
dodaj	: Liczba x Liczba	$\mapsto$ Liczba	
dziel	: Liczba x Liczba	$\rightarrow$ Liczba	(funkcja częściowa)
łącz	: Słowo x Słowo	$\mapsto$ Słowo	
twórz-li	: Dana	$\mapsto$ Lista	
połóż-na-li	: Dana x Lista	$\mapsto$ Lista	
weź-z-li	: Lista	$\rightarrow$ Dana	(funkcja częściowa)
usuń-z-li	: Lista	$\rightarrow$ Lista	(funkcja częściowa)

twórz-ta	: Dana	$\mapsto$ Tablica	
dołącz-do-ta	: Tablica x Dana	$\mapsto$ Tablica	
wymień-w-ta	: Tablica x Liczba x Dana	$\rightarrow$ Tablica	(funkcja częściowa)
weź-z-ta	: Tablica x Liczba	$\rightarrow$ Dana	(funkcja częściowa)
twórz-re	: Identyfikator x Dana	$\mapsto$ Rekord	
dołącz-do-re	: Identyfikator x Dana x Rekord	$\mapsto$ Rekord	
weź-z-re	: Rekord x Identyfikator	$\rightarrow$ Dana	(funkcja częściowa)
usuń-z-re	: Rekord x Identyfikator	$\rightarrow$ Rekord	(funkcja częściowa)
wymień-w-re	: Rekord x Identyfikator x Dana	$\rightarrow$ Rekord	(funkcja częściowa)

Zauważmy, że na tym etapie nie potrzebujemy błędów abstrakcyjnych, bo mamy do czynienia z operacjami teoretycznymi, które mogą być częściowe. Przyjmę, że konstruktory boolowskie i arytmetyczne są zdefiniowane „w zwykły sposób”. Opisane w rozdziale 2.9 boolowskie konstruktory trójwartościowe zostaną wprowadzone dopiero w algebrze kompozytów, gdzie pojawiają się błędy abstrakcyjne. Zakładam, że konstruktor **równe** jest relacją równości ograniczoną do danych prostych. To ograniczenie wynika jedynie ze względów implementacyjnych, a nie matematycznych.

Konstruktor **łącz** jest konkatencją ciągów usuwającą wewnętrzne cudzysłowy proste. Pozostałe konstruktory dają się łatwo zdefiniować za pomocą operacji na krotkach i mappingach opisanych w rozdziale 2.1. Oto ich definicje:

twórz-li.dan	= (dan)	
połącz-na-li.(dan, lis)	= lis © (dan)	
weź-z-li.lis	= szczyt.lis	
usuń-z-li.lis	= usuń.lis	
twórz-ta.dan	= [1/dan]	
dołącz-do-ta.(tab, dan)	= tab[ind/dan]	gdzie ind = max.(dom.tab) + 1
weź-z-ta.(tab, lic)	= tab.lic	

Zauważmy, że tab.lic może być nieokreślona, a wtedy nieokreślona jest również weź-z-ta.(tab, lic).

Jak też wynika z powyższych definicji, wszystkie tablice osiągalne, a więc tworzone w trakcie wykonywania programów, będą mappingami określonymi na odcinkach zbioru liczb całkowitych postaci $[1, \dots, n]$.

Ponieważ rekordy, podobnie jak tablice, są mappingami, definicje ich konstruktorów są podobne.

twórz-re.(ide, dan)	= [ide/dan]
---------------------	-------------

dołoż-do-re.(ide, dan, rek) = rek[ide/dan]

weź-z-re.(rek, ide) = rek.ide

usuń-z-re.(ide, rek) = rek[ide/?]

wymień-w-re.(rek, ide, dan) =

rek.ide = ? → ?

prawda → rek[ide/dan]

Operacja wymiany danej w rekordzie została zdefiniowana w ten sposób, aby była nieokreślona, gdy wskazany atrybut nie występuje w rekordzie. To oczywiście decyzja inżynierska, a nie matematyczna konieczność, bo rek[ide/dan] jest określona również wtedy, gdy rek.ide jest nieokreślona⁵⁴. W **Lingua** każda próba wymiany danej związanej z nieistniejącym w rekordzie atrybutem będzie prowadzić do sygnalizacji błędu.

Ze względów technicznych, których przyczyna zostanie wyjaśniona nieco później, przyjmę teraz, że *algebra danych* jest zbudowana nad dwoma nośnikami:

ide : Identyfikator

dan : Dana

a jej wszystkie operacje są traktowane jako operacje częściowe na iloczynach kartezjańskich tych nośników. Na przykład dzielenie potraktuję jako funkcję częściową:

dziel : Dana x Dana → Dana

która typologicznie jest operacją na dowolnych danych, ale jej dziedzina określoności obejmuje jedynie argumenty liczbowe, z których drugi musi być różny od zera. Z kolei operacja pobrania danej z rekordu będzie funkcją częściową:

weź-z-re : Dana x Identyfikator → Dana

nieokreślona, gdy jej pierwszy argument nie jest rekordem, a także wtedy, gdy atrybut nie występuje w rekordzie. Tak rozumiane konstruktory algebry danych będą więc wszystkie funkcjami częściowymi.

Przyjęcie dwurodzajowej algebry danych jest konsekwencją faktu, że przyszła algebra denotacji wyrażeń danologicznych będzie dwurodzajowa, co zostanie wyjaśnione pod koniec rozdziału 5.3.2.

Jak już było wspomniane, mamy więc do czynienia z nieco uogólnionym pojęciem algebry w stosunku do zdefiniowanego w rozdziale 2.11, bo tam zakładałem, że konstruktory algebry są funkcjami całkowitymi. To uogólnienie nie przeszkadza jednak, by mówić o sygnaturach takich algebr, a więc też o podobieństwie algebr, z których jedna lub obie są częściowe. Zresztą w dalszych rozważaniach algebra danych — którą oznaczę przez AlgDan — będzie jedyną algebrą o konstruktorach częściowych.

Wybór nośników i konstruktorów algebry danych jest jedną z podstawowych decyzji inżynierskich, jakie podejmujemy przy tworzeniu języka programowania, gdyż wyznacza ona kształt przyszłej części aplikatywnej języka, a w tym algebr kompozytów, korpusów, jarzm, typów, denotacji wyrażeń i wreszcie składni wyrażeń.

⁵⁴ Przyjmuję takie rozwiązanie wychodząc z założenia, że skoro programista użył operacji wymiany danej przypisanej w rekordzie wskazanemu atrybutowi, to zapewne sądził, że ten atrybut występuje w rekordzie. Jeżeli więc tak nie jest, to trzeba go o tym poinformować.

Na koniec jedna uwaga metodologiczna. Otóż dwurodzajowość algebry danych została przyjęta z myślą o budowniczym języku, ale nie pod kątem jego użytkownika, czyli programisty. Na poziomie podręcznika programisty można nadal pokazać sygnaturę wielorodzajową, jak na początku niniejszego rozdziału, i mówić o rodzajach danych, a więc o rodzaju boolowskim, liczbowym, słowowym, listowym itd. Do pojęcia *rodzaju*, jako różnego od *typu*, będę się też odwoływał na poziomie naszego modelu.

5.2.2 Algebra korpusów

Korpusy opisują „wewnętrzną strukturę danych” i posłużą do zdefiniowania pojęcia typu. Dla każdego rodzaju danych zdefiniuję odpowiadający jej rodzaj korpusów. O korpusach założę, że będą krotkami i rekordami na słowach oraz ich kombinacjami. Dziedzinę korpusów definiuję równaniem:

$$\begin{aligned} \text{kor} : \text{Korpus} = & \\ & \{('boolowska')\} \mid \{('liczba')\} \mid \{('słowo')\} \mid && (\text{korpusy proste}) \\ & \{ 'L' \} \times \text{Korpus} \mid && (\text{korpusy listowe}) \\ & \{ 'T' \} \times \text{Korpus} \mid && (\text{korpusy tablicowe}) \\ & \{ 'R' \} \times (\text{Identyfikator} \Rightarrow \text{Korpus}) && (\text{korpusy rekordowe}) \end{aligned} \quad (5.2.2-1)$$

Korpusy danych prostych są więc jednoelementowymi krotkami słów. Symbole ‘L’, ‘T’ i ‘R’ nazywam *inicjałami korpusów*. Służą one do odróżniania korpusów danych strukturalnych od korpusów ich elementów. Np. (‘T’, (liczba’)) jest korpusem tablic liczbowych, a (‘L’, (‘L’, (‘liczba’))) jest korpusem list, których elementami są tablice liczb.

W przypadku korpusów listowych, które są postaci (‘L’, kor) powiemy, że kor jest *korpusem wewnętrznym* korpusu listowego i analogicznie dla korpusów tablicowych. Elementy dziedziny

$$\text{rko} : \text{RekKor} = \text{Identyfikator} \Rightarrow \text{Korpus}$$

będę nazywał *rekordami korpusowymi*. Zatem każdy korpus rekordowy jest parą postaci (‘R’, rko).

Definicje dziedzin korpusów zapowiadają przyszłą zasadę, że wszystkie elementy list i tablic będą miały jednakowe korpusy, natomiast w przypadku rekordów, każdemu atrybutowi będzie mógł być przypisany inny korpus.

W tym miejscu należy podkreślić, że wybór korpusowych ograniczeń nakładanych na dane stanowi decyzję budowniczego języka, a nie matematyczną konieczność. Z matematycznego punktu widzenia równie dobrze mógłbym założyć, że elementy listy nie muszą mieć wspólnego korpusu, albo że wspólny korpus muszą mieć wartości atrybutów jednego rekordu. Dokonałem jednak innych wyborów, które wydają się zgodne z dość powszechną praktyką spotykaną w uniwersalnych językach programowania, pozwolą też opisać konstrukcje SQL-owe, o których będzie mowa w rozdziale 12.⁵⁵

Zauważmy, że korpus tablicowy nie określa liczby elementów tablicy. Wprowadzenie takiego ograniczenia będzie jednak możliwe za pomocą jarzm (patrz rozdział 5.2.4).

Nieco dalej każdej operacji na danych przypiszę operację na korpusach zdefiniowaną w taki sposób, aby równoległe z wyliczaniem „nowej” danej można było wyliczyć jej korpus.

⁵⁵ W denotacyjnym modelu dla SQL tabele bazy danych będą (mówiąc w pewnym uproszczeniu) listami rekordów o wspólnym korpusie, a bazy danych — rekordami tabel o różnych korpusach.

Ponieważ operacje korpusowe będą w szczególnych sytuacjach generować komunikaty błędów, wprowadzam uniwersalny zbiór błędów:

błd : Błąd

o którym założę jedynie, że tworzą go ujęte w cudzysłowy proste napisy nad jakimś ustalonym alfabetem. Przykładem błędu może być 'brak-takiego-atrybutu'. W tym momencie zbioru błędów nie muszę bliżej określać, bo definiowane dalej konstruktory będą jedynie przekazywać błąd, nie „reagując” na jego treść, ani też jej nie zmieniając. Innymi słowy zakładam w tym miejscu, że przyjęty dla **Lingua-A** system obsługi błędów jest reaktywny (patrz rozdział 2.8). Wprowadzam natomiast dziedzinę korpusów z błędami:

kor : KorpusB = Korpus | Błąd

Założę, że w *algebrze korpusów*, którą oznaczę przez AlgKor, znajdują się jedynie dwa nośniki

ide : Identyfikator

kor : KorpusB

natomiast korpusowe konstruktory zostaną dobrane w ten sposób, aby każdemu konstruktorowi (operacji teoretycznej) ope na danych odpowiadał konstruktor korpusów Kr[ope], który obliczy korpus wyniku ope na podstawie korpusów jej argumentów.

Dla powiązania danych z korpusami każdemu korpusowi przypiszę zbiór danych, o których będę mówić, że *mają dany korpus* i który nazwę *klanem* danego korpusu. Ponieważ dziedziny korpusów są rozłączne, mogę zdefiniować *funkcję klanowania korpusów* KLAN-Kr, która każdemu korpusowi przypisze jego klan:

KLAN-Kr : KorpusB $\mapsto$ Pod.Dana

Tę funkcję definiuję przez indukcję strukturalną:

KLAN-Kr.błd = $\emptyset$ dla każdego błędu błd : Błąd

KLAN-Kr.('boolowska') = Boolowska

KLAN-Kr.('liczba') = Liczba

KLAN-Kr.('słowo') = Słowo

KLAN-Kr.('L', kor) = (KLAN-Kr.kor)^{c*}

KLAN-Kr.('T', kor) = Liczba $\Rightarrow$ KLAN-Kr.kor

KLAN-Kr.('R', [ide-1/kor-1, ..., ide-n/kor-n]) =

{ [ide-1/dan-1, ..., ide-n/dan-n] | dan-i : KLAN-Kr.kor-i dla i = 1;n }

Zakładam, że klanem pustego korpusu rekordowego (tj. korpusu gdzie n = 0) jest zbiór jednoelementowy składający się z pustego rekordu i analogicznie dla tablic.

Jak widzimy, klany różnych korpusów są rozłączne, natomiast ich suma nie wyczerpuje dziedziny Dana, co oznacza, że nie każda dana ma korpus. Na przykład nie ma go lista, w której stoją obok siebie liczby, słowa i tablice. Jak zobaczymy dalej, w wyniku wykonywania wyrażeń w **Lingua-A** będą powstawać jedynie takie dane, które mają korpusy. Dla późniejszego wykorzystania korpusów przy definiowaniu konstruktorów denotacji wyrażeń wprowadzam funkcję częściową:

KOR : Dana $\rightarrow$ Korpus

która jest określona jedynie dla danych mających korpusy, i która każdej takiej danej przypisze jej korpus, a więc

dla każdego kor : Korpus, jeżeli dan : KLAN-Kr.kor to KOR.dan = kor

Na przykład:

KOR.2 = ('liczba')

KOR.[nazwisko/'Kowalski', imię/'Jan'] = ('R', [nazwisko/('słowo'), imię/('słowo')])

Ze względów technicznych, które staną się jasne nieco dalej, przyjmę też, że KOR jest określona również na identyfikatorach i że jest na nich identycznością:

KOR.ide = ide

Ponieważ klany korpusów są rozłączne, funkcja KOR jest dobrze określona.

Rozważmy teraz dowolną operację teoretyczną na danych, czyli konstruktor danych określony na danych i identyfikatorach:

ope : DanIde-1 x ... x DanIde-n $\rightarrow$ Dana

gdzie DanIde-i jest równe albo Dana albo Identyfikator. Każdej takiej operacji przypiszę transparentny (patrz rozdział 2.8) konstruktor korpusów

Kr[ope] : KorIde-1 x ... x KorIde-n $\mapsto$ KorpusB

gdzie

jeżeli DanIde-i = Identyfikator to KorIde-i = Identyfikator i na odwrót

o tej własności, że

jeżeli

ope.(arg-1,...,arg-n) jest określona **oraz**

KOR.(ope.(arg-1,...,arg-n)) jest określony **oraz**

KOR.arg-i są określone dla i = 1;n **oraz**

Kr[ope].(KOR.arg-1,...,KOR.arg-n) nie jest błędem

to

KOR.(ope.(arg-1,...,arg-n)) = Kr[ope].(KOR.arg-1,...,KOR.arg-n)

Jeżeli powyższe wyrażenie jest spełnione, to o konstruktorze Kr[ope] powiem, że jest *adekwatny dla ope*. O funkcji KOR można więc powiedzieć, że wobec konstruktorów wzajemnie adekwatnych „zachowuje się” jak częściowo-kreślony homomorfizm.

Dla każdego konstruktora algebry danych zdefiniuję teraz adekwatny dla niego konstruktor algebry korpusów. W tym celu wprowadzę funkcję pomocniczą:

rodzaj : KorpusB $\mapsto$ {'boolowska'}, ('liczba'), ('słowo'), 'L', 'T', 'R'

rodzaj.kor =

kor : Błąd $\rightarrow$ kor

kor = ('boolowska') $\rightarrow$ ('boolowska')

kor = ('liczba') $\rightarrow$ ('liczba')

kor = ('słowo') $\rightarrow$ ('słowo')

kor : {'L'} x Korpus $\rightarrow$ 'L'

kor : {'T'} x Korpus $\rightarrow$ 'T'

$\text{kor} : \{ 'R' \} \times (\text{Identyfikator} \Rightarrow \text{Korpus}) \Rightarrow 'R'$

Teraz możemy przystąpić do definiowania konstruktorów korpusów. Ich pierwsza grupa odpowiada zeroargumentowym konstruktorom identyfikatorów, tym samym co w algebrze danych, a więc w ich nazwach pomijam kontekst $\text{Kr}[\dots]$:

$\text{twórz-id.ide} : \mapsto \text{Identyfikator} \quad \text{dla ide} : \text{Identyfikator}$

Druga grupa to konstruktory korpusów, którą rozpoczynają konstruktory zeroargumentowe. Tak więc pierwsze trzy konstruktory korpusów będą następujące:

$\text{Kr}[\text{twórz-bo.bo}] : \mapsto ('boolowska') \quad \text{dla boo} : \text{Boolowska}$

$\text{Kr}[\text{twórz-lc.lic}] : \mapsto ('liczba') \quad \text{dla lic} : \text{LiczbaS}$

$\text{Kr}[\text{twórz-sł.sł}] : \mapsto ('słowo') \quad \text{dla sł} : \text{SłowoS}$

W rzeczywistości mamy tu do czynienia z trzema indeksowanymi rodzinami konstruktorów, które w granicach jednej rodziny są identyczne między sobą. Tę „algebraiczną rozrzutność” utrzymuję jedynie dla zachowania podobieństwa pomiędzy algebrami danych i korpusów, co pozwoli w przyszłości na ujednolicenie definicji kolejnych algebr. Dodawaniu przypisuję operację korpusową:

$\text{Kr}[\text{dodaj}].(\text{kor-1}, \text{kor-2}) =$

$\text{kor-i} : \text{Błąd} \quad \rightarrow \text{kor-i} \quad \text{dla } i = 1;2$

$\text{kor-1} = \text{kor-2} = ('liczba') \quad \rightarrow ('liczba')$

prawda $\rightarrow 'oczekiwana-liczba'$

i analogicznie dla pozostałych operacji na danych prostych, a w tym dla predykatu *równe*. Jest oczywiste, że wszystkie tak zdefiniowane operacje są korpusowo adekwatne. Operacje na korpusach listowych definiuję w następujący sposób:

$\text{Kr}[\text{twórz-li}].\text{kor} =$

$\text{kor} : \text{Błąd} \quad \rightarrow \text{kor}$

prawda $\rightarrow ('L', \text{kor})$

$\text{Kr}[\text{połóż-na-li}].(\text{kor-e}, \text{kor-l}) =$

(e – element, l – lista)

$\text{kor-i} : \text{Błąd} \quad \rightarrow \text{kor-i} \quad \text{dla } i = e, l$

$\text{rodzaj.kor-l} \neq 'L' \rightarrow 'oczekiwana-lista'$

niech

$('L', \text{kor-wl}) = \text{kor-l}$

(wl – wewnętrzny korpus listy)

$\text{kor-wl} \neq \text{kor-e} \rightarrow 'niezgodne-korpusy'$

prawda $\rightarrow \text{kor-l}$

$\text{Kr}[\text{weź-z-li}].\text{kor} =$

$\text{kor} : \text{Błąd} \quad \rightarrow \text{kor}$

$\text{rodzaj.kor} \neq 'L' \rightarrow 'oczekiwana-lista'$

niech

$(\text{'L'}, \text{kor-e}) = \text{kor}$
prawda $\rightarrow \text{kor-e}$

$\text{Kr}[\text{usuń-z-li}].\text{kor} =$
 $\text{kor} : \text{Błąd} \rightarrow \text{kor}$
 $\text{rodzaj.kor} \neq \text{'L'} \rightarrow \text{'oczekiwana-lista'}$
prawda $\rightarrow \text{kor}$

Operacje na korpusach tablic definiujemy podobnie:

$\text{Kr}[\text{twórz-ta}].\text{kor} =$
 $\text{kor} : \text{Błąd} \rightarrow \text{kor}$
prawda $\rightarrow (\text{'T'}, \text{kor})$

$\text{Kr}[\text{dołącz-do-ta}].(\text{kor-t}, \text{kor-d}) =$
 $\text{kor-i} : \text{Błąd} \rightarrow \text{kor-i} \quad \text{dla } i = t, d$
 $\text{rodzaj.kor-t} \neq \text{'T'} \rightarrow \text{'oczekiwana-tablica'}$
niech
 $(\text{'T'}, \text{kor-e}) = \text{kor-t}$
 $\text{kor-e} \neq \text{kor-d} \rightarrow \text{'niezgodne-korpusy'}$
prawda $\rightarrow \text{kor}$

$\text{Kr}[\text{weź-z-ta}].\text{kor} =$
 $\text{kor} : \text{Błąd} \rightarrow \text{kor}$
 $\text{rodzaj.kor} \neq \text{'T'} \rightarrow \text{'oczekiwana-tablica'}$
niech
 $(\text{'T'}, \text{kor-e}) = \text{kor}$
prawda $\rightarrow \text{kor-e}$

Ostatnią grupę stanowią operacje odpowiadające korpusom rekordowym.

$\text{Kr}[\text{twórz-re}].(\text{ide}, \text{kor}) =$
 $\text{kor} : \text{Błąd} \rightarrow \text{kor}$
prawda $\rightarrow (\text{'R'}, [\text{ide}/\text{kor}])$

$\text{Kr}[\text{dołącz-do-re}].(\text{kor-r}, \text{ide}, \text{kor-d}) =$

kor-i : Błąd → kor-i dla i = r, d

rodzaj.kor-r ≠ 'R' → 'oczekiwany-rekord'

niech

('R', rko) = kor-r

rko.ide = ! → 'atrybut-zajęty'

prawda → ('R', rek-kor[ide/kor-d])

Kr[weź-z-re].(ide, kor-r) =

kor-r : Błąd → kor-r

rodzaj.kor-r ≠ 'R' → 'oczekiwany-rekord'

niech

('R', rko) = kor-r

rko.ide = ? → 'nieznany-atrybut'

prawda → rko.ide

Kr[usuń-z-re].(ide, kor-r) =

kor-r : Błąd → kor-r

rodzaj.kor-r ≠ 'R' → 'oczekiwany-rekord'

niech

('R', rko) = kor-r

rko.ide = ? → 'nieznany-atrybut'

prawda → ('R', kor-r[ide/?])

Kr[wymień-w-re].(kor-r, ide, kor-d) =

kor-i : Błąd → kor-i dla i = r, d

rodzaj.kor-r ≠ 'R' → 'oczekiwany-rekord'

niech

('R', rko) = kor-r

rko.ide = ? → 'nieznany-atrybut'

kor-d ≠ rko.ide → 'niezgodne-korpusy'

prawda → kor-r

W tym miejscu podjąłem istotną decyzję inżynierską przyjmując, że przy wymianie danej przypisanej atrybutowi korpus nowej danej musi się pokrywać z korpusem danej poprzedniej. Ta zasada nie wynika bowiem — jak poprzednie — z definicji dziedzin korpusowych i zasady adekwatności. Zauważmy, że ta operacja zwraca albo błąd, albo wyjściowy korpus rekordowy.

Sprawdzenie, czy tak zdefiniowane operacje są korpusowo adekwatne pozostawiam czytelnikowi (mając nadzieję, że w moich definicjach nie znajdzie błędów).

5.2.3 Algebra kompozytów

Za pomocą korpusów możemy opisywać własności danych dotyczące ich „wewnętrznej struktury”. To pozwoli nam wprowadzić w **Lingua-1** zmienne deklarowane w ten sposób, aby mogły przyjmować jako swoje wartości jedynie dane o określonych korpusach, np. liczby lub słowa lub też listy rekordów określonego rodzaju itp.

Daną ustrukturyzowaną będę nazywał parę składającą się z danej i korpusu. Wprowadzam więc dziedzinę:

dus : DanUst = Dana x Korpus

O danej ustrukturyzowanej (dan, kor) powiem, że jest *dobrze ustrukturyzowana* jeżeli

dan : KLAN-Kr.kor czyli KOR.dan = kor.

Dane dobrze ustrukturyzowane będę nazywał *kompozytami*. Zatem kolejna dziedzina to:

kom : Kompozyt = {(dan, kor) | dan : KLAN-Kr.kor } (5.2.3-1)

O kompozycie (dan, kor) będę mówił, że *niesie* daną dan oraz korpus kor. Zauważmy, że kompozyty nie niosą błędów⁵⁶.

O kompozycie, który niesie daną prostą będę mówił, że jest *kompozytem prostym* i analogicznie będę rozumiał *kompozyty strukturalne*. Będę też mówili o *kompozytach boolowskich, liczbowych, listowych* itd. Obok dziedziny kompozytów wprowadzę dziedzinę kompozytów boolowskich, bo będą one odgrywały szczególną rolę:

kom : KompozytBoo = {(boo, ('boolowska')) | boo : {pp, ff}}

W przyszłości kompozyty będą wartościami wyrażeń danologicznych. Dla celów technicznych rozszerzę na kompozyty i identyfikatory wprowadzoną dla korpusów (rozdział 5.2.2) funkcję rodzaj i oznaczę ją tym samym symbolem. Tak więc:

rodzaj.(dan, kor) = rodzaj.kor

rodzaj.ide = ide

Wprowadzę też dwie nowe funkcje selekcji:

dana.(dan, kor) = dan

korpus.(dan, kor) = kor

dana.ide = ide

korpus.ide = ide

Zauważmy, że dla kompozytów prostych funkcje rodzaj i korpus się pokrywają, ale dla kompozytów strukturalnych już nie. Dziedziny kompozytów uzupełnię teraz o błędy, a więc:

⁵⁶ W tym miejscu Andrzej Tarlecki zadał mi pytanie, dlaczego wprowadzam kompozyty, skoro każda dana, która ma korpus, wyznacza ten korpus jednoznacznie przez funkcję KOR. W tej sytuacji można by operować na explicite zadanych danych i implicite wyznaczonych przez nie korpusach. Z matematycznego punktu widzenia tak oczywiście można by postąpić, zdecydowałem jednak inaczej, by pokazać explicite jak przy modyfikacji danych zmieniają się ich korpusy. Takie ujęcie sugeruje też pewną drogę budowania implementacji dla **Lingua**, a także jest moim zdaniem wygodne przy wprowadzaniu pojęcia typu i operacji na typach (rozdział 5.2.5).

kom : KompozytB = Kompozyt | Błąd

kom : KompozytBooB = KompozytBoo | Błąd

Dla kompozytów budujemy dwurodzajową *algebrę kompozytów* AlgKom podobną do AlgDan oraz AlgKor i zbudowaną nad nośnikami

ide : Identyfikator

kom : KompozytB

Każdemu konstruktorowi algebry danych ope przypiszę konstruktor kompozytów Km[ope], który na danych „wykona” ope dbając o to, aby „wysłać” do niego jedynie argumenty z obszaru jego określoności, natomiast na korpusach wykona operację Kr[ope]. Konstruktory algebry kompozytów będą więc funkcjami całkowitymi, które na obszarze nieokreśloności konstruktora danych wyemitują błąd. Te konstruktory zadbają też o to, aby generowane przez nie dane były reprezentowalne. W tym celu przyjmę, że w naszym modelu został określony uniwersalny predykat:

nadmiarowy : Kompozyt $\mapsto$ Boolowska

który wskazuje, kiedy dany kompozyt przekroczył dopuszczalną zajętość pamięci komputera⁵⁷. Tego predykatu nie definiuję explicite, gdyż będzie on zależał od implementacji. Należy jednak uwzględnić fakt, że z każdym rodzajem danych — boolowskie, liczby, słowa, listy itd. — będzie związana inna „norma gabarytowa”.

Trzeba też pamiętać, że norma gabarytowa najczęściej nie da się zapisać prostym „nie większe niż”. Na przykład norma dla liczb określi nie tylko przedział, z którego te liczby mogą pochodzić, ale też maksymalną liczbę znaków dziesiętnych dopuszczalnych w ich reprezentacji. Z tego punktu widzenia liczba $1/3$ będzie nadmiarowa, natomiast nienadmiarowa będzie np. 0,333333333333333333. Z kolei norma dla list nie może być określona jedynie przez maksymalną długości listy, gdyż zajętość pamięci przez listę zależy nie tylko od liczby jej elementów, ale również od ich gabarytów. Dla tablic i rekordów będzie podobnie.

Jedyny konkret, który przyjmę dla predykatu nadmiarowości, to założenie, że dane generowane przez konstruktory zeroargumentowe twórz-bo, twórz-lc, twórz-sł nie tworzą danych nadmiarowych. Praktycznie oznacza to, że przed wpisaniem do programu zbyt dużej liczby lub zbyt dużego słowa będzie chronić analizator składniowy. Przyjmujemy również zasadę, że

wszystkie konstruktory algebry kompozytów zostaną zdefiniowane w ten sposób, aby kompozyty osiągalne były nienadmiarowe.

Obok predykatu nadmiarowości założę też, że w modelu zdefiniowano uniwersalną funkcję zaokrąglania:

⁵⁷ Stefan Sokołowski zwrócił mi uwagę, że w niektórych zastosowaniach uwzględnianie predykatu **nadmiarowy** w dowodach całkowitej poprawności programu może prowadzić do technicznie dość złożonych rachunków, więc aby ułatwić życie programiście, może warto pomyśleć o dwuetapowym budowaniu programu: najpierw bez uwzględnienia nadmiarowości, a później dodatkowa analiza nadmiarowości. Na gruncie przedstawionego w książce modelu jest to oczywiście możliwe. Nic nie stoi na przeszkodzie, by obok „pełnej” semantyki zbudować jej wersję okrojoną przez przyjęcie, że predykat nadmiarowości jest zawsze fałszywy. Może to mieć sens nie tylko przy dwuetapowym budowaniu programów poprawnych, ale też i w takich zastosowaniach, w których wiadomo, że nadmiar „praktycznie” się nie pojawi, np. w aplikacjach finansowo-księgowych, w wielu aplikacjach bazodanowych itp. Z drugiej strony w innych zastosowaniach badanie nadmiarowości może być wysoce krytyczne. Dobrym przykładem mogą być mikroprogramy arytmetyczne o czym przekonała się w roku 1995 firma Intel, która z powodu błędu związanego z nadmiarowością w ich mikroprocesorze musiała wymienić setki tysięcy takich mikroprocesorach znajdujących się na rynku.

zaokrąglij : Dana $\mapsto$ Dana

która „przycina” liczby o za długich, a dla liczb nie wymagających przycięcia, a także dla wszystkich innych rodzajów danych, jest identycznością. Zauważmy, że bez tej funkcji wynik $1/3$ mógłby być odrzucony jako nadmiarowy.

Konstruktor na kompozytach nieboolowskich zdefiniuję jako „kartezjańskie złożenie” odpowiadających sobie konstruktorów danych i korpusów, uzupełnione o konieczne sprawdzenia (o konstruktorach dla kompozytów boolowskich nieco dalej). Niech więc:

ope : DanIde-1 x ... x DanIde-n $\mapsto$ Dana

będzie takim konstruktorem danych. Odpowiadający mu konstruktor kompozytów definiuje w sposób następujący (KomIde-i definiuje analogicznie do KorIde-i):

Km[ope] : KomIde-1 x ... x KomIde-n $\mapsto$ KompozytB

Km[ope].(arg-1,...,arg-n) =

arg-i : Błąd $\rightarrow$ arg-i dla i = 1;n

niech

dan-i = dana.arg-i dla i = 1;n

kor-i = korpus.arg-i dla i = 1;n

kor = Km[ope].(kor-1,...,kor-n)

kor : Błąd $\rightarrow$ kor (*)

ope.(dan-1,...,dan-n) = ? $\rightarrow$ komunikat (5.2.3-2)

niech

dan = zaokrąglij.(ope.(dan-1,...,dan-n))

kom = (dan, kor)

nadmiarowy.kom $\rightarrow$ 'przekroczenie-rozmiaru'

prawda $\rightarrow$ kom

Wykonanie operacji **Km[ope]** rozpoczyna się więc od sprawdzenia, czy któryś z jej argumentów nie jest błędem. Jeżeli tak jest, to pierwszy z napotkanych błędów staje się wynikiem końcowym. W ten sposób jest realizowana transparentność.

W przeciwnym przypadku następuje próba wyliczenia korpusu wynikowego i jeżeli on nie okaże się błędem, to badamy, czy wynik zastosowania konstruktora danych jest określony. Jeżeli tak nie jest, to pojawia się odpowiedni komunikat błędu. np. 'dzielenie-przez-zero'. Jeżeli te testy nie doprowadzą do błędu to jest wyliczana wynikowa dana, przez zastosowanie operacji **ope** oraz **zaokrąglij**. Przypominam, że dla danych, które nie są liczbami, **zaokrąglij** jest identycznością.

Sprawdzenie, czy wyliczany korpus nie jest błędem — klauzula (*) — realizuje zapowiedzianą w rozdziale 5.1 zasadę, że przed wykonaniem operacji na danych następuje sprawdzenie, czy te dane są dla operacji „właściwe”.

W kolejnym kroku następuje sprawdzenie, czy wynikowa dana nie przekracza dopuszczalnej wielkości. Tego kroku nie należy jednak rozumieć dosłownie w ten sposób, że najpierw tworzymy „za duży” kompozyt, a dopiero później sygnalizujemy błąd. Obecność predykatu **nadmiarowy** w naszym modelu symbolizuje fakt, że implementacja języka musi być wyposażona w mechanizm przewidywania, że nastąpi przekroczenie rozmiaru danej.

Jeżeli i ten test przebiegnie pomyślnie, to wynikowy kompozyt zostaje zaakceptowany. Jego dobre ustrukturyzowanie wynika z faktu, że operacja $Kr[o]pe$ jest adekwatna względem ope .

Tak zdefiniowana transformacja ma zastosowanie również do definiowania konstruktorów zeroargumentowych, a więc dla $n = 0$.

W tym miejscu konieczna jest uwaga metodologiczna. Otóż ogólna postać definicji funkcji $Km[o]pe$ może budzić pewną wątpliwość, gdyż z teorii obliczalności wiadomo — o czym wspomniałem w rozdziale 3.4 — że w ogólnym przypadku funkcji częściowych predykat

$$ope.(dan-1, \dots, dan-n) = ?$$

nie jest obliczalny. Jednakże w przypadku podstawowych operacji wykonywanych na danych, predykaty badające ich określoność są obliczalne i dają się łatwo zaimplementować. Na przykład przy dzieleniu sprawdzamy, czy dzielnik nie jest równy 0:

$$Km[dziel].(kom-1, kom-2) =$$

$$kom-i : \text{Błąd} \rightarrow kom-i \quad \text{dla } i = 1, 2$$

niech

$$(dan-i, kor-i) = kom-i \quad \text{dla } i = 1, 2$$

$$kor = Kr[dziel].(kor-1, kor-2)$$

$$kor : \text{Błąd} \rightarrow kor$$

$$dan-2 = 0 \rightarrow \text{'dzielenie-przez-zero'}$$

$$\textbf{prawda} \rightarrow (zaokrąglij.(dziel.(dan-1, dan-2)), kor)$$

Z opisanego wyżej schematu wynikają też definicje konstruktorów zeroargumentowych. Na przykład:

$$Km[twórz-lc.128] = (128, \text{'liczba'})$$

i analogicznie dla innych takich konstruktorów.

Wszystkie tak zdefiniowane operacje na kompozytach są transparentne względem błędów⁵⁸. Z tego właśnie powodu schemat (5.2.3-2) nie ma zastosowania do trójwartościowych operacji boolowskich McCarthy'ego (rozdział 2.9), które przyjmę jako podstawę do budowania kategorii wyrażeń boolowskich w **Lingua-A**. Konstruktor kompozytów odpowiadający koniunkcji McCarthy'ego definiuję więc w sposób bezpośredni (-K od „kompozyt”):

$$\text{oraz-K}.(kom-1, kom-2) =$$

$$kom-1 : \text{Błąd} \rightarrow kom-1$$

$$\text{rodzaj.kom-1} \neq \text{'boolowska'} \rightarrow \text{'oczekiwana-boolowska'}$$

$$\text{dana.kom-1} = ff \rightarrow (ff, \text{'boolowska'})$$

$$kom-2 : \text{Błąd} \rightarrow kom-2$$

$$\text{rodzaj.kom-2} \neq \text{'boolowska'} \rightarrow \text{'oczekiwana-boolowska'}$$

$$\textbf{prawda} \rightarrow (\text{dana.kom-2}, \text{'boolowska'})$$

⁵⁸ Trochę naciągając rozumowanie można powiedzieć, że konstruktory zeroargumentowe są również transparentne, bo „jeśli otrzymają na wejściu kompozyty będące błędami, to zwrócą błąd”, a więc ich warunek transparentności jest spełniony tożsamościowo, bo przesłanka tego warunku jest zawsze fałszywa.

Zauważmy, że skoro doszło do sprawdzenie przedostatniej klauzuli, to można stąd wnioskować, że $\text{dana.kom-1} = \text{pp}$, a zatem dana.kom-2 jest taka jak w dana.kom-1 . Tak zdefiniowana operacja jest oczywiście korpusowo adekwatna.

Zwracam uwagę, że tym razem nie posłużyłem się notacją Km[oraz] , gdyż oraz-K nie odwołuje się w żaden sposób do konstruktora klasycznego oraz .

Konstruktor dla negacji jest oczywisty:

$\text{nie-K.kom} =$

$\text{kom} : \text{Błąd}$	$\rightarrow \text{kom}$
$\text{rodzaj.kom} \neq (\text{'boolowska'})$	$\rightarrow \text{'oczekiwana-boolowska'}$
$\text{kom} = (\text{pp}, (\text{'boolowska'}))$	$\rightarrow (\text{ff}, (\text{'boolowska'}))$
$\text{kom} = (\text{ff}, (\text{'boolowska'}))$	$\rightarrow (\text{pp}, (\text{'boolowska'}))$

Konstruktor dla alternatywy definiuję tak, aby były spełnione prawa De Morgana, a więc:

$\text{lub-K.}(\text{kom-1}, \text{kom-2}) =$
 $\text{nie-K.}(\text{oraz-K.}(\text{nie-K.kom-1}, \text{nie-K.kom-2}))$

5.2.4 Algebra transferów

Posługując się pojęciem korpusu możemy wyrażać te cechy danych, które w wielu językach programowania wyczerpują pojęcie typu, np. typ boolowski, liczbowy, listowy, tablicowy itd. Istnieją jednak języki, które oferują większą ekspresywność typów. Na przykład, w standardzie SQL możemy zadeklarować typy tabel, w których kolumnom przypisano typy odnoszące się nie tylko do korpusów danych, ale też do innych cech danych, np. *small number*, a niekiedy i do całej kolumny np. *unique* (bez powtórzeń). Typ tabeli może też dodatkowo określać predykat globalny, który ma być spełniony przez każdy jej wiersz. Z kolei typ bazy danych może nieść informację o relacjach nadrzędności pomiędzy tabelami.

By móc budować tak bogate typy w językach z serii **Lingua**, trzeba wprowadzić konstrukcje, które pozwolą na budowanie predykatów opisujących własności kompozytów. To z kolei wymaga wprowadzenia szerszego pojęcia transferu.

Transferem nazwę każdą jednoargumentową funkcję przekształcającą kompozyty (i błędy) w kompozyty (i błędy). Wprowadzam więc dziedzinę:

$\text{tra} : \text{Transfer} = \text{KompozytB} \mapsto \text{KompozytB}$

Szczególnym rodzajem transferu będą *jarzma* odwzorowujące kompozyty w kompozyty boolowskie. Ich dziedziną będzie więc:

$\text{jar} : \text{Jarzmo} = \text{KompozytB} \mapsto \text{KompozytBooB}$

Konstruktory kompozytów zostaną zdefiniowane w ten sposób, aby wszystkie transfery osiągalne były transparentne względem błędów, a więc spełniały warunek

$\text{tra.błd} = \text{błd}$ dla każdego $\text{błd} : \text{Błąd}$

Powiem, że *kompozyt kom spełnia tranzyt (jarzmo) tra*, jeżeli:

$\text{tra.kom} = (\text{pp}, (\text{'boolowska'}))$

Jarzma są więc jednoargumentowymi predykatami na kompozytach. Antycypując przyszłą składnię języka, wyrażeniu jarzmowemu

value < 10

odpowiada jarzmo, które będzie spełnione, gdy kompozyt zadany mu jako argument jest liczbowy i niesie liczbę mniejszą niż 10. W tym wyrażeniu **value** nie jest identyfikatorem zmiennej, ale słowem kluczowym reprezentującym daną niesioną przez argument kompozytowy. Innym przykładem wyrażenia jarzmowego może być

value + 2 < 10

wyrażające fakt, że wartość danej w aktualnym kompozycie powiększona o dwa jest mniejsza od 10. Denotacyjnie to jarzmo będzie złożeniem poprzedniego jarzma z transferem, który zwiększa niesioną przez kompozyt daną o 2. Z kolei wyrażenie

record.pensja + **record.premia** < 7000

odpowiada jarzmu, które jest spełnione, gdy kompozyt zadany mu jako argument niesie rekord o atrybutach liczbowych **pensja** i **premia**, których suma wartości jest mniejsza od 7000⁵⁹.

Analogicznie jak w przypadku danych, korpusów i kompozytów zbuduję teraz dwurodzajową algebrę transferów AlgTra:

ide : Identyfikator

tra : Transfer

Zauważmy, że nośniki algebry transferów nie zawierają błędów abstrakcyjnych, zawierają natomiast transfery (a w tym jarzma), które zwracają błędy. Jest to też algebra z zupełnie innego poziomu niż poprzednie, gdyż wśród jej elementów znajdują się transfery, a więc konstruktory algebry kompozytów.

Większość konstruktorów transferów będzie — podobnie jak konstruktory korpusów i kompozytów — pochodnymi operacji na danych, choć niekoniecznie jedynie tych operacji, które występują w algebrze danych, a także niekoniecznie wszystkich tych operacji. Przez **Kt[o]** oznaczę konstruktor transferów przypisany operacji teoretycznej **ope**.

Obok operacji na danych opisanych w rozdziale 5.2.1 do budowania transferów przyjmę dodatkowe operacje. By nie wydłużać zbytnio naszych rozważań, niech będą to:

suma	: Liczba ^{c+}	↦ Liczba	— suma liczb wchodzących w skład listy
maks	: Liczba ^{c+}	↦ Liczba	— maksymalna liczba na liście
mała-lc	: Liczba	↦ Boolowska	— np. liczba z przedziału [-9999, 9999]
rosnąco-lc	: Liczba ^{c+}	↦ Boolowska	— lista liczb uporządkowana rosnąco

Pierwsze dwie nazywa się w SQL *operacjami agregowania*. Trzecia to typowy predykat opisujący typ pola SQL-owej tabeli. Czwarty wprowadzam dla pokazania, że w naszym modelu możemy w definiowaniu jarzm, a więc i typów pójść nieco dalej niż w typowych językach programowania.

Fakt, że ww. operacje nazwałem „dodatkowymi” oznacza, że nie wejdą one do repertuaru wyrażań „zwykłych”, a będą dostępne jedynie przy tworzeniu transferów. To oczywiście (jak już nieraz) nie matematyczna konieczność, ale wybór budowniczego języka przyjęty w tym przypadku zgodnie ze standardem SQL. A oto lista konstruktorów algebry AlgTra podzielona na kilka grup:

⁵⁹ Z matematycznego punktu widzenia moglibyśmy jarzma zapisywać na poziomie składni bez słów kluczowych, a więc np. „< 10” lub „+2<10”, taka składnia byłaby jednak mocno nieintuicyjna.

Konstruktory identyfikatorów

twórz-id.ide : $\mapsto$ Identyfikator dla ide : Identyfikator

Konstruktory transferów przetwarzających dane proste

Kt[twórz-lic.lic]	:	$\mapsto$ Transfer	dla lic : LiczbaS
Kt[twórz-sł.sło]	:	$\mapsto$ Transfer	dla słó : SłowoS
Kt[dodaj]	: Transfer x Transfer	$\mapsto$ Transfer	
Kt[dziel]	: Transfer x Transfer	$\mapsto$ Transfer	
Kt[suma]	: Transfer	$\mapsto$ Transfer	
Kt[maks]	: Transfer	$\mapsto$ Transfer	

Konstruktory jarzm

Kt[równe]	: Transfer x Transfer	$\mapsto$ Transfer	
Kt[mniejsze]	: Transfer x Transfer	$\mapsto$ Transfer	
Kt[mała-lic]	: Transfer	$\mapsto$ Transfer	
Kt[rosnąco-lic]	: Transfer	$\mapsto$ Transfer	
Kt[twórz-bo.boó]	:	$\mapsto$ Transfer	dla boó : Boolowska
oraz-T	: Transfer x Transfer	$\mapsto$ Transfer	
lub-T	: Transfer x Transfer	$\mapsto$ Transfer	
nie-T	: Transfer	$\mapsto$ Transfer	

Konstruktory jarzm kwantyfikatorowych

wszystkie-na-li	: Transfer	$\mapsto$ Transfer
wszystkie-w-ta	: Transfer	$\mapsto$ Transfer

Konstruktory transferów selekcji z list, tablic i rekordów

Kt[weź-z-li]	:	$\mapsto$ Transfer
Kt[weź-z-ta]	: Transfer	$\mapsto$ Transfer
Kt[weź-z-re]	: Identyfikator	$\mapsto$ Transfer

Konstruktor identycznościowy

przełącz	:	$\mapsto$ Transfer
----------	---	--------------------

Konstruktory identyfikatorów są nam już znane z poprzednich algebr.

Dwa kolejne konstruktory tworzą transfery o wartościach stałych (poza błędami). W przypadku danych liczbowych każdej liczbie lic odpowiada inny konstruktor, a więc np.:

$Kt[tw\acute{o}rz-lc.2] : \mapsto Transfer$
 $Kt[tw\acute{o}rz-lc.2].().kom =$
 $kom : B\acute{ł}ad \rightarrow kom$
prawda $\rightarrow Km[tw\acute{o}rz-lc.2].()$

Tę definicję można też zapisać w postaci wyraźniej, tj. bez odwoływania się do konstruktora kompozytów:

$Kt[tw\acute{o}rz-lc.2].().kom =$
 $kom : B\acute{ł}ad \rightarrow kom$
prawda $\rightarrow (2, ('liczba'))$

Zeroargumentowe konstruktory odpowiadające słowom i danym boolowskim definiuję analogicznie. Każdy z tych konstruktorów tworzy transfer, którego wartością jest ustalony kompozyt prosty niezależny od kompozytu „na wejściu”, chyba że na wejściu będzie błąd.

Kolejne cztery konstruktory transferów, które odpowiadają operacjom arytmetycznym, definiuję wg. jednolitego schematu: dla każdego konstruktora danych ope definiuję odpowiadający mu konstruktor transferów $Kt[ope]$:

$$Kt[ope].(tra-1, \dots, tra-n).kom = Km[ope].(tra-1.kom, \dots, tra-n.kom) \quad (5.2.4-1)$$

Ten schemat przyjmuję dla wszystkich (mogących się pojawić w przyszłości) konstruktorów danych prostych. Zauważmy, że jeżeli wszystkie tra-i są transparentne względem błędów oraz $Km[ope]$ też ma tę własność, to i $Kt[ope].(tra-1, \dots, tra-n)$ jest transparentny.

Pierwsze cztery konstruktory jarzm definiuję zgodnie ze schematem (5.2.4-1), pod który podpadają też dwa konstruktory zeroargumentowe dla boo : Boolowska:

$$Kt[tw\acute{o}rz-bo.boo].() = Km[tw\acute{o}rz-bo.boo].() = (boo, ('boolowska'))$$

W tym miejscu wprowadzam dwa oznaczenia:

$$PP = Kt[tw\acute{o}rz-bo.pp]$$

$$FF = Kt[tw\acute{o}rz-bo.ff]$$

Dla pozostałych operacji boolowskich przyjmę konstruktory zgodne z rachunkiem zdań Kleene'ego (por. rozdział 2.9), a nie McCarthy'ego, jak w przypadku kompozytów. W przypadku koniunkcji taka definicja wygląda w sposób następujący (-T od „transfer”):

$$\text{oraz-T.}(tra-1, tra-2).kom =$$

$$kom : B\acute{ł}ad \rightarrow kom$$

niech

$$kom-i = tra-i.kom \quad \text{dla } i = 1;2$$

$$kom-i = (ff, ('boolowska')) \rightarrow (ff, ('boolowska')) \quad \text{dla } i = 1;2$$

$$kom-i : B\acute{ł}ad \rightarrow kom-i \quad \text{dla } i = 1;2$$

$$\text{rodzaj.kom-i} \neq ('boolowska') \rightarrow \text{'oczekiwana-boolowska'} \quad \text{dla } i = 1;2$$

prawda $\rightarrow$ (pp, ('boolowska'))

Zatem dla sfalsyfikowania tej koniunkcji wystarczy, aby dowolny z jej argumentów niósł ff. Jeżeli tak nie jest, to wynikiem wykonania koniunkcji jest albo błąd, albo kompozyt niosący pp. Konstruktor nie-T definiuję jak u McCarthy'ego, a lub-T tak, aby były spełnione prawa De Morgana.

Zwracam uwagę, że w przypadku konstruktorów dla kompozytów przyjąłem rachunek McCarthy'ego bo tam — jak zobaczymy w dalszych rozdziałach — wyrażenia będą mogły generować obliczenia nieskończone, a to dzięki procedurom funkcyjnym (rozdział 7.5). Jednakże, ponieważ procedur funkcyjnych do transferów nie wprowadzę, mogę przyjąć dla nich „bardziej leniwy” rachunek Kleene'ego. Ten właśnie rachunek został przyjęty w standardzie SQL (rozdział 12).

Zauważmy jednakże, że jarzmo tworzone przez oraz-T jest — zgodnie z ogólnym założeniem o naszych jarzmach — transparentne względem błędów. Leniwa ewaluacja dotyczy jedynie błędów generowanych przez transfery składowe tra-1 i tra-2. Oczywiście podobną własność ma jarzmowa alternatywa.

Nazwy konstruktorów boolowskich nie mają formy Kt[opec], gdyż nie odwołują się do konstruktorów algebry danych, ale są tworzone ad hoc. Te samą cechę mają dwa poniższe konstruktory transferów dla list i tablic odpowiadające kwantyfikowaniu ogólnemu. Podobnie jak konstruktory boolowskie, te poniższe też tworzą jarzma z jarzm, formalnie rzecz biorąc stosują się jednak do dowolnych transferów.

wszystkie-na-li : Transfer $\mapsto$ Transfer (wszystkie na liście)

wszystkie-na-li.tra.kom =

kom : Błąd $\rightarrow$ kom
rodzaj.kom $\neq$ 'L' $\rightarrow$ 'oczekiwana-lista'

niech

(dan-1,...,dan-n) = dana.kom
('L', kor) = korpus.kom (elementy listy mają wspólny korpus)

kom-i = tra.(dan-i, kor) dla i = 1;n

kom-i : Błąd $\rightarrow$ kom-i dla i = 1;n

($\forall$ i = 1 ; n) kom-i = (pp, ('boolowska')) $\rightarrow$ (pp, ('boolowska'))

prawda $\rightarrow$ (ff, ('boolowska'))

Po tej wstępnej obróbce kompozytu na wejściu sprawdzamy, czy wszystkie elementy niesionej przez niego listy spełniają transfer tra. W analogiczny sposób definiuję konstruktor jarzm dla tablic:

wszystkie-w-ta : Transfer $\mapsto$ Transfer

wszystkie-w-ta.tra.kom =

kom : Błąd $\rightarrow$ kom

rodzaj.kom $\neq$ 'T' $\rightarrow$ 'oczekiwana-tablica'

niech

[lic-1/dan-1,...,lic-n/dan-n] = dana.kom

('T', kor) = korpus.kom (elem. tablicy mają wspólny korpus)

kom-i = tra.(dan-i, kor) dla $i = 1;n$

kom-i : Błąd $\rightarrow$ kom-i

($\forall i = 1;n$) kom-i = (pp, ('boolowska')) $\rightarrow$ (pp, ('boolowska'))

prawda $\rightarrow$ (ff, ('boolowska'))

Zauważmy, że jeżeli tablica jest wielowymiarowa, to zbudowane jarzmo odnosi się do elementów tablicy pierwszego poziomu. Tę konstrukcję można jednak iterować.

Kolejne trzy konstruktory odpowiadają operacjom selekcji związanym z danymi strukturalnymi. Te konstruktory są pochodnymi operacji na danych, co ma wpływ na ich nazwy. Pierwszy tworzy transfer pobierający szczytowy element listy:

Kt[weż-z-li] : $\mapsto$ Transfer

Kt[weż-z-li].() = Km[weż-z-li]

czyli:

Kt[weż-z-li].().kom = Km[weż-z-li].kom

gdzie Km[weż-z-li] jest zdefiniowane zgodnie ze schematem (5.2.3-2). Ten konstruktor tworzy więc transfer, który jest konstruktorem algebry kompozytów. Oczywiście Km zadba o sprawdzenie, czy kom jest kompozytem listowym, a także, czy nie jest błędem.

Zauważmy, że definicja konstruktora Kt[weż-z-li] nie podpada pod schemat (5.2.4-1) dlatego, że kompozyt listowy kom przekazywany do konstruktora Km[weż-z-li] nie jest przetwarzany przez żaden transfer. To ograniczenie jest ponownie wynikiem decyzji inżynierskiej oznaczającej, że

jarzma nie będą „wewnętrznie” modyfikowały kompozytów strukturalnych.

Jedynie dopuszczalne w naszym modelu przetworzenie kompozytu strukturalnego przez transfer to selekcja elementu. To samo ograniczenie dotyczy obu kolejnych konstruktorów. Z tego właśnie powodu wszystkie te konstruktory mają o jeden argument mniej niż odpowiadające im konstruktory danych. Np. Km[weż-z-li] pobiera jeden argument, którym jest kompozyt listowy i w związku z tym Kt[weż-z-li] jest zeroargumentowy.

Przypominam, że w przypadku danych prostych takie ograniczenie nie zostało wprowadzone, co ilustrowało wyrażenie jarzmowe **value** + 2 < 10. Odpowiadający mu transfer najpierw modyfikuje wejściowy kompozyt operacją dodawania, a dopiero później wykonuje sprawdzenie. W naszej przyszłej składni wyrażeń transferowych nie da się natomiast opisać transferu, który na przykład najpierw modyfikuje zadany rekord przez usunięcie z niego jakiegoś atrybutu, a następnie wykonuje sprawdzenie:

record.pensja + **record.premia** < 7000

Drugi z tej grupy konstruktorów tworzy transfer pobierający element tablicy wskazany przez zadany indeks, który jest wyliczany przez jego jedyny argument:

Kt[weż-z-ta] : Transfer $\mapsto$ Transfer

$$\text{Kt}[\text{weż-z-ta}].\text{tra.kom} = \text{Km}[\text{weż-z-ta}].(\text{kom}, \text{tra.kom})$$

Ponownie za wszystkie sprawdzenia odpowiada konstruktor kompozytów $\text{Km}[\text{weż-z-ta}]$, który sprawdza, czy kom niesie tablicę, a tra.kom —liczbę. On również generuje błąd, gdy kom jest błędem.

Zauważmy, że ponieważ kom musi nieść tablicę, a transfer tra musi generować liczbę, to przy obecnym repertuarze konstruktorów transferów, tra może być jedynie transferem o stałej wartości liczbowej, a więc powstającym w wyniku zastosowania zeroargumentowego konstruktora

$$\text{Kt}[\text{twórz-lc.lic}]$$

Gdybyśmy jednak wśród transferów tablic umieścili transfer licz-ele-ta zwracający liczbę elementów tablicy, to wtedy transfer

$$\text{Kt}[\text{weż-z-ta}].\text{licz-ele}$$

zwracałby ostatni element tablicy.

Trzeci konstruktor w sposób analogiczny do drugiego tworzy transfer „pobierający” kompozyt z rekordu:

$$\text{Kt}[\text{weż-z-re}] : \text{Identyfikator} \mapsto \text{Transfer}$$

$$\text{Kt}[\text{weż-z-re}].\text{ide.kom} = \text{Km}[\text{weż-z-re}].(\text{ide}, \text{kom})$$

Ostatni konstruktor transferów **przekaż** nie odpowiada żadnej operacji na danych, więc jego nazwa ponownie nie ma kontekstu $\text{Kt}[\dots]$. Ten zero-argumentowy konstruktor tworzy transfer identycznościowy:

$$\text{przekaż}().\text{kom} = \text{kom}$$

Służy on do tego, aby w schemacie definicji (5.2.4-1) niektóre z argumentów $\text{Km}[\text{ope}]$ mogły być kompozytem wejściowym kom , a nie przetworzonym tra-i.kom . Jeżeli na poziomie składni konkretnej **przekaż** zapiszemy jako **value** (weź wartość), to wyrażenie jarzmore

$$\text{value} < 10,$$

odpowiada zgodnie ze schematem (5.2.4-1) jarzmu

$$\text{Kt}[\text{mniejsze-lc}].(\text{przekaż}(), \text{Kt}[\text{twórz-lc.10}].()).$$

Rozwijając to wyrażenie przy założeniu, że $\text{kom} = (\text{lic}, ('liczba'))$ otrzymamy:

$$\begin{aligned} &\text{Kt}[\text{mniejsze-lc}].(\text{przekaż}(), \text{Kt}[\text{twórz-lc.10}].()).\text{kom} = \\ &\quad \text{Km}[\text{mniejsze-lc}].(\text{przekaż}().\text{kom}, \text{Km}[\text{twórz-lc-10}].().\text{kom}) = \\ &\quad \text{Km}[\text{mniejsze-lc}].(\text{kom}, (10, ('liczba'))) = \\ &\quad (\text{mniejsze-lc}(\text{lic}, 10), ('boolowska')) \end{aligned}$$

Teraz, podobnie jak dla korpusów, również dla transferów wprowadzam operator klanowania. *Klanem transferu* nazwę zbiór kompozytów, dla których ten transfer jest spełniony:

$$\text{KLAN-Tr} : \text{Transfer} \mapsto \text{Pod.Kompozyt}$$

$$\text{KLAN-Tr.tra} = \{\text{kom} \mid \text{tra.kom} = (\text{pp}, ('boolowska'))\}$$

Oczywiście klany transferów nieboolowskich będą puste⁶⁰.

⁶⁰ W związku z tym można by zadać pytanie, dlaczego definiujemy klany dla transferów, a nie jedynie dla jarzm? To pytanie ma szerszy kontekst, a mianowicie, dlaczego w ogóle wprowadzamy transfery,

Rozważmy teraz przytaczany już przykład wyrażenia jarzmowego opisującego własność rekordu

```
record.pensja + record.premia < 7000
```

Odpowiadające mu jarzmo to transfer:

```
Kt[mniejsze].(  
  Kt[dodaj].(Kt[weż-z-re].(twórz-id.pensja.())), Kt[weż-z-re].(twórz-id.premia.()) ),  
  Kt[twórz-lc].7000.() )
```

Możemy też opisać własność listy rekordów, której wszystkie elementy mają mieć tę własność:

```
all-list(record.pensja + record.premia < 7000)
```

Jemu odpowiada jarzmo

```
wszystkie-na-li.(  
  Kt[mniejsze].(  
    Kt[dodaj].(Kt[weż-z-re].(twórz-id.pensja.())), Kt[weż-z-re].(twórz-id.premia.()) ),  
    Kt[twórz-lc].7000.()  
  )  
)
```

Możemy też połączyć powyższą własność lokalną listy z jej własnością globalną:

```
all-list(record.pensja + record.premia < 7000) and  
sum(record.premia) < 100.000
```

Oczywiście dla zapisania treści tego jarzma trzeba by jeszcze wprowadzić funkcję sumowania wartości wskazanego atrybutu rekordu po wszystkich elementach listy rekordów.

Na koniec jedna uwaga metodologiczna. Otóż wszystkie zdefiniowane w tym rozdziale konstruktory transferów tworzą transfery, które opisując własności kompozytów odnoszą się „głównie” do własności zawartych w nich danych, a co do korpusów ograniczają się do sprawdzenia, czy są one odpowiedniego rodzaju. Np. transfer budowany konstruktorem **wszystkie-na-li** sprawdza, czy zadany mu kompozyt niesie listę. To ograniczenie to oczywiście wybór inżynierski, a nie konieczność matematyczna. Przyjąłem je, by nie komplikować zbyt naszego modelu, a także dlatego, że transfery są w **Lingua** istotnie wykorzystywane dopiero przy budowaniu struktur bazodanowych typu SQL-owego, gdzie transfery odnoszą się do korpusów jedynie w taki sposób, jaki został opisany w niniejszym rozdziale.

5.2.5 Algebra typów

Typem będę nazywał parę składającą się z korpusu i transferu. Wprowadzam zatem dziedzinę⁶¹:

skoro tak naprawdę interesują nas jedynie jarzma? Odpowiedź wiąże się z faktem, że nasz model opieramy na algebrach, a algebra jarzm bez uwzględnienia w niej transferów byłaby bardzo uboga. Podobnie uboga byłaby algebra denotacji wyrażeń boolowskich, gdybyśmy nie mogli „korzystać” w niej z denotacji wyrażeń nieboolowskich. Z kolei, jak zobaczymy w rozdziale 5.2.5, algebra transferów wraz z algebrą korpusów posłużą do zdefiniowania algebry typów. Skoro więc z transferów nie mogę zrezygnować, to wygodniej mi będzie zdefiniować funkcję KLAN-Tr ogólnie dla dowolnych transferów.

⁶¹ Dlaczego tu znów dowolne transfery, a nie jarzma, starałem się wyjaśnić w przypisie z rozdziału 5.2.4 dotyczącym funkcji KLAN-Tr.

$\text{typ} : \text{Typ} = \text{Korpus} \times \text{Transfer}$

O typie (kor, tra) powiem, że *nie*sie korpus kor oraz transfer (jarzmo) tra. Powiem, o nim, że jest *boolowski*, *liczbowy*, *słowny* itd., gdy kor jest odpowiedniego rodzaju. W podobny sposób określe pojęcia *typu prostego* i *typu strukturalnego*. O typie, którego jarzmem jest PP powiem, że jest *typem bezjarzmowym*. Zauważmy, że typy — podobnie jak kompozyty — nie niosą błędów.

Z każdym typem związę zbiór kompozytów tego typu, który nazwę *klanem typu*. Definiuję więc funkcję:

$\text{KLAN-Ty} : \text{Typ} \mapsto \text{Pod.Kompozyt}$

$\text{KLAN-Ty}(\text{kor}, \text{tra}) =$

$\{(\text{dan}, \text{kor}) \mid \text{dan} : \text{KLAN-Kr.kor} \text{ **oraz** } (\text{dan}, \text{kor}) : \text{KLAN-Tr.tra}\}$

O typie, którego klan jest pusty powiem, że jest *typem pustym*. Zbuduję teraz *algebrę typów* o trzech nośnikach:

$\text{ide} : \text{Identyfikator}$

$\text{tra} : \text{Transfer} = \text{KompozytB} \mapsto \text{KompozytB}$

$\text{typ} : \text{TypB} = \text{Typ} \mid \text{Błąd}$

która stanie się fundamentem dla przyszłej algebry denotacji wyrażeń typologicznych. Do konstruktorów algebry typów zaliczę:

1. wszystkie konstruktory identyfikatorów,
2. wszystkie konstruktory algebry transferów,
3. niżej zdefiniowane konstruktory typów.

Wiele konstruktorów typów, podobnie jak konstruktory kompozytów, będzie się odwoływać do konstruktorów korpusów i będzie pochodnymi konstruktorów danych. Przez $\text{Ky}[\text{o}pe]$ oznaczę konstruktor typów odwołujący się do konstruktora danych $\text{o}pe$. Jednakże w odróżnieniu do konstruktorów kompozytów, przy budowaniu których wykorzystałem wszystkie operacje na danych, w przypadku typów wykorzystam jedynie te, które prowadzą do utworzenia nowego typu. Wykorzystam więc na przykład konstruktor tworzenia list, ale już nie wykorzystam dodawania nowego elementu do listy, gdyż ta operacja nie zmienia typu.

A oto ta część sygnatury algebry typów, która jest ograniczona do konstruktorów trzeciej grupy i którą podzielę z kolei na trzy podgrupy:

$\text{Ky}[\text{twórz-bo}].\text{boo}$	:	$\mapsto \text{TypB}$	dla boo : Boolowska
$\text{Ky}[\text{twórz-lc}].\text{lic}$	:	$\mapsto \text{TypB}$	dla lic : Liczba
$\text{Ky}[\text{twórz-sł}].\text{sło}$	:	$\mapsto \text{TypB}$	dla sło : Słowo

$\text{Ky}[\text{twórz-li}]$	:	TypB	$\mapsto \text{TypB}$
$\text{Ky}[\text{twórz-ta}]$	:	TypB	$\mapsto \text{TypB}$
$\text{Ky}[\text{twórz-re}]$	:	$\text{TypB} \times \text{Identyfikator}$	$\mapsto \text{TypB}$
$\text{Ky}[\text{dołoż-do-re}]$	:	$\text{TypB} \times \text{Identyfikator} \times \text{TypB}$	$\mapsto \text{TypB}$

$$\text{Ky}[\text{usuń-z-re}] : \text{TypB} \times \text{Identyfikator} \mapsto \text{TypB}$$

$$\text{wymień-ty-tr} : \text{TypB} \times \text{Transfer} \mapsto \text{TypB}$$

Nazwa ostatniego z tych konstruktorów nie występuje w kontekście $\text{Ky}[\dots]$ ponieważ ten konstruktor nie odpowiada żadnej operacji na danych. Jak zobaczymy dalej, dokonuje on wymiany transferu w typie. Konstruktory pierwszej grupy służą do tworzenia prostych typów bezjarzmowych:

$$\text{Ky}[\text{twórz-bo.boó}].() = (\text{Kr}[\text{twórz-bo.boó}].(), \text{PP}) = ((\text{boo}, ('boolowska')), \text{PP})$$

$$\text{Ky}[\text{twórz-lc.lic}].() = (\text{Kr}[\text{twórz-lc.lic}].(), \text{PP}) = ((\text{lic}, ('liczba')), \text{PP})$$

$$\text{Ky}[\text{twórz-sł.sło}].() = (\text{Kr}[\text{twórz-sł.sło}].(), \text{PP}) = ((\text{sło}, ('słowo')), \text{PP})$$

Konstruktory drugiej podgrupy odwołują się do konstruktorów danych i konstruktorów jarzm:

Tworzenie typu listowego

$$\text{Ky}[\text{twórz-li}] : \text{TypB} \mapsto \text{TypB}$$

$$\text{Ky}[\text{twórz-li}].\text{typ} =$$

$$\text{typ} : \text{Błąd} \rightarrow \text{typ}$$

niech

$$(\text{kor}, \text{tra}) = \text{typ}$$

$$\text{nowy-kor} = \text{Kr}[\text{twórz-li}].\text{kor}$$

$$\text{nowy-tra} = \text{wszystkie-na-li.tra}$$

$$\text{nowy-kor} : \text{Błąd} \rightarrow \text{nowy-kor}$$

$$\text{prawda} \rightarrow (\text{nowy-kor}, \text{nowy-tra})$$

Typ wynikowy odpowiada listom, których wszystkie elementy są typu „wejściowego”, a więc mają wspólny korpus ('L', kor) oraz spełniają wspólne jarzmo tra. Oczywiście jeżeli tra nie jest jarzmem to taki typ będzie pusty. Typ tablicowy tworzymy w sposób analogiczny:

Tworzenie typu tablicowego

$$\text{Ky}[\text{twórz-ta}] : \text{TypB} \mapsto \text{TypB}$$

$$\text{Ky}[\text{twórz-ta}].\text{typ} =$$

$$\text{typ} : \text{Błąd} \rightarrow \text{typ}$$

niech

$$(\text{kor}, \text{tra}) = \text{typ}$$

$$\text{nowy-kor} = \text{Kr}[\text{twórz-ta}].\text{kor}$$

nowy-tra = wszystkie-w-ta.tra
nowy-kor : Błąd $\rightarrow$ nowy-kor
prawda $\rightarrow$ (nowy-kor, nowy-tra)

Tworzenie typu rekordowego z jednym atrybutem

$Ky[\text{twórz-re}] : \text{TypB} \times \text{Identyfikator} \mapsto \text{TypB}$

$Ky[\text{twórz-re}].(\text{typ}, \text{ide}) =$

typ : Błąd $\rightarrow$ typ

niech

(kor, tra) = typ

nowy-kor = $Kr[\text{twórz-re}].(\text{ide}, \text{kor})$

nowy-tra = $Kt[\text{weź-z-re}].\text{ide} \bullet \text{tra}$

nowy-kor : Błąd $\rightarrow$ nowy-kor

prawda $\rightarrow$ (nowy-kor, nowy-tra)

Tak utworzony typ ma korpus [ide/kor] oraz transfer nowy-tra, który jest spełniony, gdy:

1. kompozyt wejściowy dla niego jest rekordowy i niesie atrybut ide,
2. związany z tym atrybutem kompozyt spełnia transfer tra.

Zauważmy, że transfer $Kt[\text{weź-z-re}].\text{ide}$ dokonuje selekcji kompozytu przypisanego do atrybutu ide, a kompozyt jest przekazywany (operacja $\bullet$) do transferu tra. Zatem wartość jedyne go atrybutu powstałego rekordu ma spełniać transfer wskazany przez typ będący argumentem konstruktora.

Dłożenie nowego atrybutu do typu rekordowego

$Ky[\text{dołoż-do-re}] : \text{TypB} \times \text{Identyfikator} \times \text{TypB} \mapsto \text{TypB}$

$Ky[\text{dołoż-do-re}].(\text{typ-r}, \text{ide}, \text{typ-d}) =$ (r – rekordowy, d – dokładany)

typ-i : Błąd $\rightarrow$ typ-i dla $i = r, d$

niech

(kor-i, tra-i) = typ-i dla $i = r, d$

nowy-kor = $Kr[\text{dołoż-do-re}].(\text{kor-r}, \text{ide}, \text{kor-d})$

nowy-tra = $\text{oraz-T}.\text{tra-r}, Kt[\text{weź-z-re}].\text{ide} \bullet \text{tra-i}$

nowy-kor : Błąd $\rightarrow$ nowy-kor

prawda $\rightarrow$ (nowy-kor, nowy-tra)

Korpus nowego typu jest tworzony za pomocą konstruktora korpusów, który jest odpowiedzialny za sprawdzenie, czy korpus kor-r jest rekordowy, a identyfikator ide nie jest zajęty. Nowe jarzmo stanowi, że nowy rekord spełnia zastane jarzmo tra-r na zastanych atrybutach

oraz nowe jarzmo jam-d na nowym atrybucie ide (por. komentarz do konstrukcji Kt[twórz-re].ide • tra).

Usunięcie atrybutu z typu rekordowego

$Ky[usuń-z-re] : TypB \times Identyfikator \mapsto TypB$

$Ky[usuń-z-re].(typ, ide) =$

$typ : Błąd \rightarrow typ$

niech

$(kor, jar) = typ$

$nowy-kor = Kr[usuń-z-re].(ide, kor)$

$nowy-kor : Błąd \rightarrow nowy-kor$

prawda $\rightarrow (nowy-kor, jar)$

Usunięcie atrybutu z typu rekordowego usuwa atrybut z korpusu, ale nie zmienia jarzma. Oczywiście po takiej zmianie jarzmo może nie być spełnione, o ile odwoływało się do usuniętego atrybutu. Aby uniknąć takiej sytuacji należy posłużyć się konstruktorem wymiany jarzma (ogólnie transferu), by odpowiednio je zmodyfikować przed usunięciem atrybutu.

Wymiana transferu w typie

$wymień-ty-tr : TypB \times Transfer \mapsto TypB$

$wymień-ty-tr.(typ, tra) =$

$typ : Błąd \rightarrow typ$

niech

$(kor, tra-z) = typ$ (z – zastany)

prawda $\rightarrow (kor, tra)$

Ten konstruktor dokonuje wymiany zastanego transferu na nowy. To oczywiście bardzo ogólna operacja, więc dla celów praktycznych należałoby pomyśleć o konstruktorach bardziej specyficznych, np. dopisujących koniunktywnie jarzmo do jarzma. Tego jednak nie robię, by na tym etapie budowy języka nie wchodzić zbyt głęboko w szczegóły techniczne. Zauważmy też, że jest to jedyny konstruktor algebry typów, który zmienia transfer typu pozostawiając korpus niezmienny. Jest to też jedyny konstruktor, który powoduje, że algebra typów musi zawierać wśród swoich nośników dziedzinę transferów.

5.3 Algebra denotacji wyrażeń

5.3.1 Wartości i stany pamięci

Jak już wskazywałem w rozdziale 4.1, aby móc opisywać funkcje będące denotacjami przyszłych wyrażeń, instrukcji i deklaracji, trzeba zdefiniować dziedzinę stanów pamięci. W

prostym języku programowania można by przyjąć, że są nimi wartościowania, o których była mowa w tantym rozdziale. Odpowiadałoby to sytuacji, w której pamięć operacyjna programu przechowuje jedynie dane powiązane z identyfikatorami. Jednakże w większości języków programowania występują mechanizmy wymagające, aby w pamięci składować dane wraz z ich typami, a także procedury i typy (niezależnie), a to wymaga bogatszego pojęcia stanu.

W naszym modelu stany będą wiązać z identyfikatorami danologicznymi *dane utypowane* składające się z danej lub *pseudodanej* Ω oraz typu. Zatem:

$$\text{dut} : \text{DanUty} = (\text{Dana} \mid \{\Omega\}) \times \text{Typ}$$

Na daną utypowaną $(\text{dan}, (\text{kor}, \text{jar}))$ można też spojrzeć jako na parę $((\text{dan}, \text{kor}), \text{jar})$ składającą się z kompozytu (dan, kor) i jarzma, a także jako na trójkę $(\text{dan}, \text{kor}, \text{jar})$. W przyszłości będę korzystał wymiennie z każdej z tych postaci. Zauważmy, że dane utypowane nie przechowują błędów. O danej utypowanej (dan, typ) powiem, że jest *dobrze utypowana*, gdy:

- $\text{dan} = \Omega$ lub
- $\text{dan} : \text{KLAN-Ty.typ}$.

Dane dobrze utypowane będę nazywał *wartościami*. Wprowadzam więc dziedzinę:

$$\text{war} : \text{Wartość} = \{(\text{dan}, \text{typ}) \mid \text{dan} = \Omega \text{ lub } \text{dan} : \text{KLAN-Ty.typ}\}$$

Parę postaci (Ω, typ) nazywam *pseudowartością*, a kompozyt postaci (Ω, kor) — *pseudokompozytem*. Pseudowartości będą przypisywane identyfikatorom danologicznym w chwili ich deklarowania. Wartość, która nie jest pseudowartością, nazwę *wartością właściwą*. Na pseudokompozyty rozszerzam funkcję identyfikacji rodzaju:

$$\text{rodzaj}(\Omega, \text{kor}) = \Omega$$

co oznacza też, że rozszerzam też przeciwdziedzinę tej funkcji dodając do niej Ω . Rozszerzam ją też na wartości:

$$\text{rodzaj}(\text{kom}, \text{jar}) = \text{rodzaj.kom}$$

Stany pamięci, w skrócie *stany*, będą przechowywać:

- wartości przypisane zmiennym danologicznym,
- typy przypisane zmiennym typologicznym,
- procedury i funkcje przypisane ich identyfikatorom.

Dziedzinę stanów definiuję w sposób następujący:

$$\text{sta} : \text{Stan} = \text{Środ} \times \text{Skład} \quad (\text{stan})$$

$$\text{śro} : \text{Środ} = \text{ŚroTyp} \times \text{ŚroPro} \quad (\text{środowisko})$$

$$\text{skł} : \text{Skład} = \text{Wart} \times (\text{Błąd} \mid \{\text{'OK'}\}) \quad (\text{skład})$$

$$\text{wrt} : \text{Wart} = \text{Identyfikator} \Rightarrow \text{Wartość} \quad (\text{wartościowanie})$$

$$\text{śrt} : \text{ŚroTyp} = \text{Identyfikator} \Rightarrow \text{Typ} \quad (\text{środowisko typów})$$

$$\text{śrp} : \text{ŚroPro} = \text{Identyfikator} \Rightarrow \text{Procedura} \quad (\text{środowisko procedur})$$

Podział stanu na dwie dwuelementowe składowe w miejsce jednej czteroelementowej nie jest przypadkowy. Jak zobaczymy w dalszych rozdziałach, znajdzie on swoje uzasadnienie na gruncie modelu dla procedur (rozdział 7). Tam też zostanie zdefiniowana dziedzina *Procedura*.

Skład to ta część stanu, która w *wartościowaniu* przechowuje wartości wiążąc je z identyfikatorami. Zasadę, by wartościowania przechowywały jedynie dane dobrze utypowane, zostanie zrealizowania w mechanizmach instrukcji przypisania oraz przekazywania parametrów do procedury.

Komunikat błędu, gdy zostanie wygenerowany, staje się elementem składu, a następnie jest przekazywany wszystkim kolejnym stanom wykonania programu. Jednakże dopóki tak się nie stanie, skład niesie komunikat 'OK', który oznacza, że błąd się nie pojawił. Jeżeli komunikat jest różny od 'OK', to mówimy, że *stan (skład) niesie błąd*. Ponownie zakładam, że zbiór błędów zawiera wszystkie komunikaty, które pojawią się w przyszłych definicjach wszystkich konstruktorów denotacji. Oczywiście zadbam też o to, aby wszystkie denotacje imperatywne, tj. przekształcające stany w stany, były transparentne względem błędów w tym sensie, że nie zmieniają stanu, który niesie błąd, a wszystkie denotacje aplikatywne, tj. przekształcające stany w kompozyty generowały błąd, gdy stan niesie błąd⁶².

Środowisko to ta część stanu, gdzie są przechowywane zdefiniowane przez użytkownika typy i procedury, a w tym i procedury funkcyjne.

Dla opisanego mechanizmu błędów na poziomie stanów wprowadzam cztery funkcje pomocnicze:

$\text{błąd} : \text{Stan} \mapsto \text{Błąd} \mid \{\text{'OK'}\}$ (operator selekcji błędu)

$\text{błąd}.\text{(śro, (wrt, bld))} = \text{bld}$

$\text{jest-błąd} : \text{Stan} \mapsto \text{Boolowska}$ (predykat wykrywania błędu w stanie)

$\text{jest-błąd.sta} =$

$\text{błąd.sta} \neq \text{'OK'} \rightarrow \text{pp}$

prawda $\rightarrow \text{ff}$

$\text{jest-błąd} : \text{Skład} \mapsto \text{Boolowska}$ (predykat wykrywania błędu w składzie)

$\text{jest-błąd}.\text{(wrt, bld)} =$

$\text{bld} \neq \text{'OK'} \rightarrow \text{pp}$

prawda $\rightarrow \text{ff}$

$\blacktriangleleft : \text{Stan} \times \text{Błąd} \mapsto \text{Stan}$ (operator wprowadzania komunikatu błędu)

$\text{(śro, (wrt, bld))} \blacktriangleleft \text{bld-1} =$

$\text{(śro, (wrt, bld-1))}$

Wprowadzam też operator, który posłuży do zapewnienia, że identyfikator zadeklarowany w wartościowaniu nie może być jednocześnie zadeklarowany w środowisku i na odwrót. Denotacyjnie taka sytuacja byłaby dopuszczalna, gdyż — jak zobaczymy dalej — każde „sięgnięcie”

⁶² Oczywiście ta zasada będzie utrzymana jedynie do czasu, gdy wprowadzimy mechanizmy obsługi błędów (rozdział 12.7.6.4).

do identyfikatora będzie wskazywało komponent stanu, do którego należy sięgnąć, jednakże uznałem, że takie rozwiązanie może sprzyjać błędom programistycznym.

zadeklarowany : Identyfikator $\mapsto$ Stan $\mapsto$ BoolowskaB

zadeklarowany.ide.((śrt, śrp), (wrt, błd)) =

błd $\neq$ 'OK' $\rightarrow$ błd

śrt.ide = ! **lub** śrp.ide = ! **lub** wrt.ide = ! $\rightarrow$ pp

prawda $\rightarrow$ ff

Predykat zadeklarowany jest spełniony dla danego identyfikatora w danym stanie, który nie niesie błędu, jeżeli ten identyfikator został związany w tym stanie z typem, procedurą lub wartością.

5.3.2 Denotacje wyrażeń danologicznych

Wyrażenia danologiczne to „zwykłe” wyrażenia, odpowiadają więc one funkcjom odwzorowującym stany w kompozyty lub błędy. Dziedzina ich denotacji stanowi jeden z nośników przyszej algebry denotacji naszego języka:

dwd : DenWyrDan = Stan $\rightarrow$ Kompozyt | Błąd

Należy podkreślić, że wynikami wyrażeń danologicznych mogą być jedynie kompozyty, a więc nie pseudokompozyty. Zauważmy też, że denotacje wyrażeń danologicznych są funkcjami częściowymi na stanach, gdyż w przyszłości (rozdział 7.5) wyrażenie będą mogły zawierać procedury funkcyjne, które mogą generować obliczenia nieskończone.

Ponieważ problem stopu programu jest nierozstrzygalny (por. rozdział 3.4), nie możemy założyć, że w przypadku zagrożenia niekończącym się obliczeniem pojawi się komunikat błędu. W konsekwencji musimy założyć, że w takiej sytuacji wartość denotacji wykonywanej dla danego stanu będzie nieokreślona.

W tym miejscu należy wyjaśnić, dlaczego denotacje wyrażeń danologicznych odwzorowują stany w kompozyty, a nie w wartości. Otóż wybrałem takie rozwiązanie, gdyż najczęściej to co podlega przetwarzaniu przy wyliczaniu wyniku wykonania wyrażenia, to dana i jej korpus, a więc kompozyt. Jarzma są związane z identyfikatorami zmiennych, a ich rolą jest zapewnienie, że przypisywany zmiennej kompozyt będzie spełniał wskazane jarzmo. Z zasady jarzma nie są więc przekształcane w trakcie wyliczania wartości wyrażeń.

Denotację wyrażenia danologicznego dwd nazwę *transparentną* ze względu na błędy niesione przez stany, jeżeli:

dwd.(śro, (wrt, błd)) = błd gdy błd $\neq$ 'OK'

O konstruktorze denotacji wyrażeń danologicznych powiem, że jest *rzetelny*, jeżeli przekształca denotacje transparentne w transparentne. Wszystkie konstruktory algebry denotacji wyrażeń danologicznych zostaną zdefiniowane tak, aby były rzetelne.

W przyszłości obok denotacji wyrażeń przekształcających stany w kompozyty, błędy i typy, o których powiem, że są *denotacjami aplikatywnymi*, będziemy mieli również denotacje instrukcji, deklaracji zmiennych i definicji typów, które przekształcają stany w stany i o których powiem, że są *denotacjami imperatywnymi*.

Klasę konstruktorów denotacji wyrażeń danologicznych można podzielić na trzy kategorie:

1. konstruktor denotacji wyrażeń składających się z jednego identyfikatora; takie wyrażenia będę nazywał *zmiennymi*,
2. konstruktory denotacji wyrażeń pochodne konstruktorom kompozytów,
3. konstruktor denotacji wyrażeń warunkowych.

Pierwszy konstruktor tworzy *denotacje zmiennych*:

zmienna-dan : Identyfikator $\mapsto$ DenWyrDan

zmienna-dan.ide.sta =

jest-błąd.sta $\rightarrow$ błąd.sta

niech

(śro, (wrt, 'OK')) = sta

wrt.ide = ? $\rightarrow$ 'zmienna-niezadeklarowana'

niech

((dan, kor), jar) = wrt.ide

dan = Ω $\rightarrow$ 'zmienna niezainicjalizowana'

prawda $\rightarrow$ (dan, kor)

Wyliczenie wartości zmiennej rozpoczyna się od sprawdzenia, czy identyfikator zmiennej *ide* został zadeklarowany oraz zainicjalizowany. Jeżeli tak się nie stało, to zostaje wyemitowany odpowiedni komunikat błędu. W przeciwnym przypadku kompozyt przypisany w wartościowaniu identyfikatorowi *ide* staje się wynikiem wykonania wyrażenia. Zauważmy, że jarzmo zostaje pominięte, gdyż wynikami wykonania wyrażeń są kompozyty. Przypisane zmiennym jarzma znajdują zastosowanie dopiero w instrukcji przypisania (rozdział 6.1.4).

Implementacyjnie pojawienie się komunikatu błędu oznacza jego wyświetlenie i przerwanie biegu programu. Zauważmy też, że w ten sposób

eliminujemy ewentualną pseudowartość z dalszych obliczeń, co oznacza, że nigdy nie zostanie „wysłana” do konstruktora kompozytów jako argument.

Konstruktory drugiej grupy są definiowane po jednym dla każdego konstruktora kompozytów. Rozpocznę od konstruktorów kompozytów transparentnych względem błędów, a więc konstruktorów nieboolowskich. Niech:

$Km[o_{pe}] : KomIde-1 \times \dots \times KomIde-n \mapsto KompozytB$

będzie takim konstruktorem dla $n \geq 0$. Odpowiadający mu konstruktor denotacji wyrażeń dąnologicznych, który oznaczam przez $Kdd[Km[o_{pe}]]$, jest zdefiniowany poniższym schematem, gdzie oczywiście $DenWyrDanIde-i$ to albo $DenWyrDan$ albo $Identyfikator$.

$Kdd[Km[o_{pe}]] : DenWyrDanIde-1 \times \dots \times DenWyrDanIde-n \mapsto DenWyrDan$

$Kdd[Km[o_{pe}]].(arg-1, \dots, arg-n).sta =$ (5.3.2-1)

jest-błąd.sta $\rightarrow$ błąd.sta

arg-i.sta = ? $\rightarrow$? dla arg-i spoza dziedziny Identyfikator

niech

kom-i = dla $i = 1;n$

$\text{arg-i} : \text{Identyfikator} \rightarrow \text{arg-i}$
 $\text{prawda} \rightarrow \text{arg-i.sta}$
 $\text{prawda} \rightarrow \text{Km[ope].(kom-1, \dots, kom-n)}$

Jeżeli stan niesie błąd, to wykonanie programu zostaje przerwane i ten błąd staje się wynikiem końcowym obliczenia, co oznacza, że zostanie wyświetlony na monitorze.

W przeciwnym przypadku podejmowana jest próba wyliczenia wartości każdego z wyrażeń składowych, które nie jest identyfikatorem. W tym miejscu należy podkreślić, że wyrażenie będące zmienną, nie jest identyfikatorem w powyższym sensie, ale wyrażeniem. W schemacie (5.3.2-1) identyfikator „jako taki”, a więc nie w postaci denotacji wyrażenia zmiennad_{an.ide}, może się pojawić jedynie jako atrybut rekordu.

Jeżeli którakolwiek z tych prób doprowadzi do obliczenia nieskończonego, to nie zakończy się również wyliczenie wartości wyrażenia głównego. W tym miejscu podejmujemy decyzję inżynierską czyniąc konstruktor $\text{Kdd}[\text{Km[ope]}]$ „transparentny” nie tylko względem błędów, ale też względem nieskończonych obliczeń. Jak możemy się spodziewać, w przypadku operacji boolowskich tak nie będzie.

Jeżeli żadne z tych obliczeń nie będzie nieskończone, to uzyskane tą drogą kompozyty, identyfikatory lub błędy zostają „wysłane” do konstruktora kompozytów. Jego transparentność zapewni transparentność konstruktora denotacji. Oczywiście $\text{Km[ope].(kom-1, \dots, kom-n)}$ może być komunikatem błędu.

Zauważmy teraz, że gdybyśmy schemat (5.3.2-1) zastosowali do „leniwych” konstruktorów boolowskich, to byłyby one leniwe jedynie względem błędów. Ponieważ jednak chcemy, aby były leniwe również względem nieskończonych obliczeń, musimy napisać dla nich odrębne definicje. Zauważmy, że w tym przypadku nazwa konstruktora denotacji nie jest pochodną nazwy konstruktora kompozytów, bo nie odwołuje się do tego konstruktora:

$\text{oraz-dwd} : \text{DenWyrDan} \times \text{DenWyrDan} \mapsto \text{DenWyrDan}$

$\text{oraz-dwd}(\text{dwd-1}, \text{dwd-2}).\text{sta} =$ (5.3.2-2)

$\text{jest-błąd.sta} \rightarrow \text{błąd.sta}$

$\text{dwd-1.sta} = ? \rightarrow ?$

niech

$\text{kom-1} = \text{dwd-1.sta}$

$\text{kom-1} : \text{Błąd} \rightarrow \text{kom-1}$

niech

$(\text{dan-1}, \text{kor-1}) = \text{dwd-1.sta}$

$\text{kor-1} \neq (\text{'boolowska'}) \rightarrow \text{'oczekiwana-boolowska'}$

$\text{dan-1} = \text{ff} \rightarrow \text{ff}$ (*)

$\text{dwd-2.sta} = ? \rightarrow ?$

niech

$\text{kom-2} = \text{dwd-2.sta}$

$\text{kom-2} : \text{Błąd} \rightarrow \text{kom-2}$

niech

(dan-2, kor-2) = kom-2

kor-2 ≠ ('boolowska') → 'oczekiwana-boolowska'

prawda → (('boolowska'), dan-2)

Zauważmy, że wyliczenie wartości wyrażenia zbudowanego nad oraz-dwd rozpoczynamy od próby wyliczenia wartości pierwszego argumentu i jeżeli ono zakończy się wartością ff, to przerwamy wyliczanie i ff staje się wartością całego wyrażenia (klauszula (*)). W ten sposób unikamy wyliczenia wartości drugiego argumentu, w szczególności więc ewentualnie niekończącego się obliczenia lub sygnału błędu. Konstruktory dla pozostałych operacji Boolowskich tworzymy analogicznie.

Jedyny konstruktor trzeciej grupy odpowiada wyrażeniom warunkowy⁶³:

gdy : DenWyrDan x DenWyrDan x DenWyrDan ↦ DenWyrDan

gdy.(dwd-1, dwd-2, dwd-3).sta =

jest-błąd.sta → błąd.sta

dwd-1.sta = ? → ?

niech

kom-1 = dwd-1.sta

kom-1 : Błąd → kom-1

niech

(dan-1, kor-1) = kom-1

kor-1 ≠ ('boolowska') → 'oczekiwana-wartość-boolowska'

dan-1 : Błąd → dan-1

dan-1 = pp → dwd-2.sta

dan-1 = ff → dwd-3.sta

Dla skrócenia zapisu przyjmuję, że dwie ostatnie klauzule obejmują również przypadek, gdy obliczenie dwd-2.sta lub dwd-3.sta się nie kończy. W tym przypadku mamy do czynienia z leniwą ewaluacją, gdyż przy wyliczaniu wartości dwd-2.sta nie interesujemy się, ani czy ewaluacja dwd-3 nie prowadzi do błędu, ani też czy się kończy⁶⁴ i analogicznie dla dwd-3.

5.3.3 Postać wyraźna definicji konstruktorów denotacji

Z punktu widzenia budowniczego implementacji języka schemat (5.3.2-1) definicji konstruktora Kdd[Km[opec]] można potraktować jako deklarację wchodzącej w skład interpretera procedury, która wywołuje procedurę Km[opec], a ta z kolei wywołuje procedury opec oraz zaokrąglij (schemat (5.2.3-2)). Taki tryb postępowania odpowiada budowaniu interpretera w konwencji wstępującej (ang. bottom-up). Z punktu widzenia użytkownika języka czytelniejsze mogą być jednak definicje konstruktorów w postaci wyraźnej, które odwołują się jedynie do

⁶³ Ten konstruktor nazywam **gdy** ponieważ **jeżeli** rezerwuję dla instrukcji warunkowych.

⁶⁴ Przyjęcie w tym miejscu zasady ewaluacji leniwej jest istotną decyzją konstruktora języka, gdyż pozwala na użycie w wyrażeniach funkcji częściowych bez zagrożenia sygnałami błędu. Zauważmy bowiem, że jeżeli przez $\text{sqr}(x)$ oznaczmy pierwiastek kwadratowy z liczby x , to wyrażenie **if** $x > 0$ **then** $\text{sqr}(x)$ **else** $\text{sqr}(-x)$ **fi** wykonywane w trybie ewaluacji ochoczej wygeneruje sygnał błędu dla każdego x różnego od zera.

operacji *ope* oraz *zaokrąglj*. A oto przykład takiej definicji dla konstruktora dopisywania elementu do listy, gdzie *-dw* oznacza, że mamy do czynienia z konstruktorem denotacji wyrażeń:

połóż-na-li-dw.(*dwd-l*, *dwd-e*).*sta* = (l – lista, e – element)

jest-błąd.sta → *błąd.sta*

dwd-i.sta = ? → ? dla *i* = l, e

dwd-i.sta : Błąd → *dwd-i.sta* dla *i* = l, e

niech

(*lis*, *kor-l*) = *dwd-l.sta*

(*ele*, *kor-e*) = *dwd-e.sta*

rodzaj.kor-l ≠ 'L' → 'oczekiwana-lista'

niech

('L', *kor-wl*) = *kor-l*

kor-wl ≠ *kor-e* → 'niezgodne-korpusy'

niech

nowa-lista = *połóż-na-li*.(*lis*, *ele*)

nowy-kom-lis = (*nowa-lista*, *kor-l*)

nadmiarowy.nowy-kom-lis → 'przekroczenie-zakresu'

prawda → *nowy-kom-lis*

5.3.4 Denotacje wyrażeń typologicznych

Denotacje wyrażeń typologicznych to funkcje całkowite odwzorowujące stany w typy lub błędy:

$\text{dwt} : \text{DenWyrTyp} = \text{Stan} \mapsto \text{TypB}$

Znajdą one zastosowanie w definicjach stałych typologicznych (rozdział 6.1.3), deklaracjach zmiennych danologicznych (rozdział 6.1.2) i deklaracjach procedur (rozdział 7.3.4). W szczególności pozwolą na kaskadowe definiowanie typów przez odwoływanie się do typów zdefiniowanych wcześniej i zapisanych w środowisku typów.

Denotacje wyrażeń typologicznych są podobne do denotacji wyrażeń danologicznych, jednakże tym razem identyfikatory odwołują się do środowisk typów, a nie do wartościowań, a ponadto są to funkcje całkowite

Definicje konstruktorów denotacji rozpoczną od konstruktora *stałej typologicznej*. W tym przypadku mówimy o stałej, a nie o zmiennej, gdyż raz ustanowione wartości stałych typologicznych nie są już w programie zmieniane.

$\text{stała-typ} : \text{Identyfikator} \mapsto \text{DenWyrTyp}$

stała-typ.sta =

jest-błąd.sta → *błąd.sta*

niech

((*śrt*, *śrp*), *skł*) = *sta*

śrt.ide = ? ➔ 'stała-typologiczna-niezdefiniowana'

prawda ➔ śrt.ide

Zauważmy, że w odróżnieniu od zmiennej danologicznej, tym razem nie jest przewidziana sytuacja, gdy zmienna została zadeklarowana, ale nie zainicjalizowana. Wynika to z faktu, że definicje stałych typologicznych (rozdział 6.1.3) zawsze przypisują stałym konkretny typ.

Pozostałe konstruktory z tej grupy budujemy odwołując się do konstruktorów algebry typów w podobny sposób, jak miało to miejsce dla wyrażeń danologicznych, gdzie odwoływaliśmy się do konstruktorów kompozytów. Kdt oznacza konstruktor, który z konstruktorów typologicznych i transferowych tworzy konstruktory denotacji.

Kdt[Ky[twórz-bo.bo0]].().sta =

jest-błąd.sta ➔ błąd.sta

prawda ➔ Ky[twórz-bo.bo0]

i oczywiście analogicznie dla stałych liczbowych i słowowych. Konstruktor denotacji wyrażeń tworzących typy listowe ma następującą definicję:

Kdt[Ky[twórz-li]] : DenWyrTyp $\mapsto$ DenWyrTyp

Kdt[Ky[twórz-li]].dwt.sta

jest-błąd.sta ➔ błąd.sta

niech

typ = dwt.sta

typ : Błąd ➔ typ

prawda ➔ Ky[twórz-li].typ

Tak więc typ, który ma być użyty do zbudowania typu listowego, jest wyliczany ze stanu, a następnie stosujemy do niego odpowiedni konstruktor algebry typów. W analogiczny sposób definiujemy pozostałe konstruktory.

Kdt[Ky[twórz-ta]] : DenWyrTyp $\mapsto$ DenWyrTyp

Kdt[Ky[twórz-re]] : DenWyrTyp x Identyfikator $\mapsto$ DenWyrTyp

Kdt[Ky[dołóż-do-re]] : DenWyrTyp x Identyfikator x DenWyrTyp $\mapsto$ DenWyrTyp

Kdt[wymień-tr] : DenWyrTyp x Transfer $\mapsto$ DenWyrTyp

Podobnie jak w przypadku konstruktorów denotacji danologicznych, również i dla konstruktorów denotacji typologicznych możemy pisać ich definicje w postaci wyrażnej.

Zauważmy, że ostatni z tych konstruktorów nie odwołuje się do operacji na danych, ale bezpośrednio do konstruktora typów, a jednym z jego argumentów jest transfer.

5.3.5 Algebra denotacji wyrażeń danologicznych, typologicznych i transferowych

Na algebrę denotacji wyrażeń danologicznych, typologicznych i transferowych, którą będziemy nazywać *algebrą denotacji wyrażeń* i oznaczymy przez AlgDenWyr, składają się cztery nośniki:

ide : Identyfikator

dwd : DenWyrDan = Stan $\rightarrow$ KompozytB

tra : DenWyrTra = Transfer

dwt : DenWyrTyp = Stan $\mapsto$ TypB

oraz związane w tymi nośnikami konstruktory zdefiniowane w poprzednich rozdziałach.

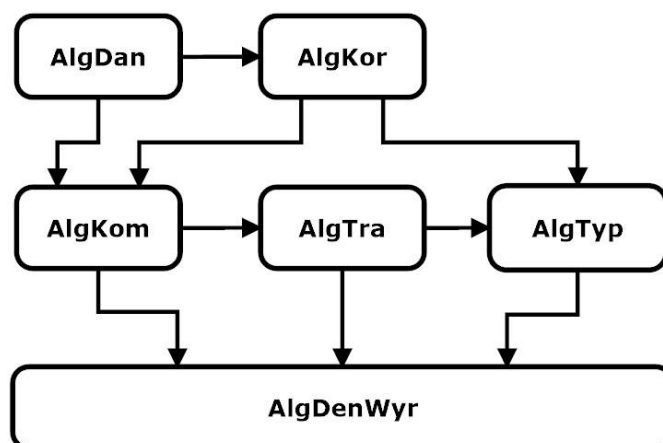
Fakt, że denotacjami wyrażeń transferowych są transfery, a nie funkcje ze stanów w transfery wynika oczywiście z faktu, że w modelu dla stanów nie uwzględniłem możliwości „zapamiętywania” transferów przez przypisywanie ich do identyfikatorów, jak to jest w przypadku danych i typów. Jak już nieraz to oczywiście decyzja inżynierska, a nie matematyczna konieczność. Przyjąłem takie rozwiązanie jedynie po to, aby nie komplikować zbytnio naszego modelu.

W tym miejscu mogę wyjaśnić, dlaczego przyjąłem, że algebra danych jest dwurodzajowa, co spowodowało, że dwurodzajowe były też algebry korpusów, kompozytów i wartości. Otóż powód tej decyzji leży dopiero w algebrze denotacji i wiąże się z faktem, że gdyby w tej algebrze chcieć wprowadzić odrębne nośniki dla denotacji wyrażeń liczbowych, Boolowskich, słowowych, listowych itd., to dla każdego z tych nośników trzeba by zdefiniować odrębny konstruktor zmiennej, co na poziomie składni oznaczałoby, że identyfikatory zmiennych musiałyby być jakoś etykietowane rodzajami. Takie rozwiązanie byłoby oczywiście możliwe, ale raczej niepraktyczne i chyba nigdzie nie stosowane. Skoro więc zdecydowałem, że na poziomie algebry denotacji nie rozróżniamy pomiędzy rodzajami danych, to nie było powodu przyjmować, że w leżących u jej podstaw algebrach danych miałyby być inaczej.

5.3.6 Sześć kroków do algebry denotacji wyrażeń

Opisany w rozdziałach 5.2 oraz 5.3 standardowy sposób przechodzenia od algebry danych do algebry denotacji wyrażeń danologicznych określam jako *podnoszenie algebr danych na poziom algebry denotacji wyrażeń*. Podsumujmy tę drogę (patrz też Rys. 5.3-1):

1. Rozpoczynamy od zdefiniowania dwurodzajowej algebry danych AlgDan. Mimo że występują w niej dane różnych rodzajów, to zostają one wszystkie połączone w jeden rodzaj Dana. Ta decyzja wynika z faktu, że w algebrze denotacji wyrażeń danologicznych mamy tylko jeden rodzaj denotacji takich wyrażeń (wyjaśnienie tej decyzji na końcu rozdziału 5.3.5)
2. Następnie definiujemy podobną do niej algebrę korpusów AlgKor, której konstruktory są adekwatne do odpowiadających im konstruktorów danych, tj. tworzą korpusy odzwierciedlające strukturę danych.
3. Nad tymi dwiema algebrami budujemy podobną do nich algebrę kompozytów AlgKom będących parami składającymi się z danej i jej korpusu. Dla każdej operacji na danych definiujemy jednoznacznie przypisaną jej operację na kompozytach.
4. Nad algebrą kompozytów budujemy (niepodobną do niej) dwurodzajową algebrę transferów AlgTra będących funkcjami przekształcającymi kompozyty w kompozyty. Szczególnym rodzajem kompozytów są jarzma przyjmujące jako wartości kompozyty boolowskie.
5. Nad algebrami korpusów i transferów (jarzm) budujemy trzyrodzajową algebrę typów AlgTyp będących parami składającymi się z korpusu i jarzma.



Rys. 5.3-1 Sześć algebr od danych do denotacji

6. Nad algebrami kompozytów, transferów i typów budujemy czterorodzajową algebrę denotacji wyrażeń danologicznych, typologicznych i transferowych oraz identyfikatorów nazwaną AlgDenWyr. Konstruktory denotacji wyrażeń danologicznych odwołują się do konstruktorów kompozytów, a konstruktory denotacji wyrażeń typologicznych — do konstruktorów typów. Do tych konstruktorów dodajemy konstruktory zmiennych danologicznych i stałych typologicznych i być może jeszcze jakieś konstruktory nie odwołujące się do wcześniej zbudowanych algebra, jak w naszym przypadku gdy. Ponieważ denotacje wyrażeń transferowych są po prostu transferami, ich konstruktory przenoszą się do algebry denotacji bez żadnych zmian.

Kroki 1, 2 i 4 mają charakter kreatywny. Tu podejmujemy podstawowe decyzje dotyczące aplikatywnej części języka. Pozostałe kroki mają charakter dość standardowy, co powinno pozwolić na ich — przynajmniej częściowe — zalgorytmizowanie.

5.4 Algebry składni

5.4.1 Składnia abstrakcyjna języka Lingua-A

W rozdziale 4.5 opisałem pięć kroków tworzenia modelu denotacyjnego aplikatywnej części języka programowania. W tej chwili więc, po zbudowaniu algebry denotacji wyrażeń, mamy za sobą pierwszy krok na drodze do zdefiniowania **Lingua-A**, a więc aplikatywnej części języków z grupy **Lingua**. Teraz należy zbudować składnię abstrakcyjną, konkretną i kolokwialną dla **Lingua-A**.

Jak wiemy z rozdziału 2.12, wychodząc z sygnatury algebry denotacji możemy „algorytmicznie wygenerować” gramatykę równaniową składni abstrakcyjnej. To zadanie wykonamy dla algebry denotacji wyrażeń AlgDenWyr tworząc odpowiadającą jej algebrę składni abstrakcyjnej AlgWyrA. Każdemu z czterech nośników denotacyjnych przypiszemy odpowiadający mu nośnik składniowy (Tab. 5.4-1). Przyrostek A oznacza przynależność do składni abstrakcyjnej.

denotacje	składnie	opis
Identyfikator	Identyfikator	identyfikatory
DenWyrDan	WyrDanA	wyrażenia danologiczne

Transfer	WyrTraA	wyrażenie transferowe
DenWyrTyp	WyrTypA	wyrażenia typologiczne

Tab. 5.4-1 Nośniki algebr związanych z wyrażeniami

Gramatykę składni abstrakcyjnej zapiszę przy użyciu skrótów notacyjnych wprowadzonych w rozdziale 2.14. Każdej z kategorii składniowych odpowiada jedno równanie dziedzinowe w gramatyce algebry składni. Pierwsze równanie opisuje dziedzinę identyfikatorów:

Identyfikatory

ide : Identyfikator =

{twórz-id.ide.() | ide : Identyfikator}

Napisanie tego równania w standardowej notacji dla gramatyk składni abstrakcyjnej jak w rozdziale 2.12 prowadziło do konieczności wylistowania explicite wszystkich dopuszczalnych identyfikatorów. Zamiast tego na poziomie formalnym stosuję powyższy skrót notacyjny, a na poziomie podręcznika użytkownika wyjaśnię jakie znaki mogą wystąpić w identyfikatorach i ile ich może być. Na poziomie implementacji powstanie prosty program sprawdzający, czy w identyfikatorze nie użyto niedozwolonych symboli oraz czy identyfikator nie zawiera ich zbyt wiele.

Wyrażenia danologiczne

wyd : WyrDanA =

stale

{Kdd[Km[twórz-bo.bo.()]] | boo : Boolowska} |
 {Kdd[Km[twórz-lc.lic.()]] | lic : LiczbaS} |
 {Kdd[Km[twórz-sł.sło.()]] | sł : SłowoS} |

zmienne

zmienna-dan.(Identyfikator) |

wyrażenia boolowskie

oraz-dwd (WyrDanA , WyrDanA) |
 lub-dwd (WyrDanA , WyrDanA) |
 nie-dwd (WyrDanA) |
 Kdd[Km[mniejsze]] (WyrDanA , WyrDanA) |

wyrażenia arytmetyczne

Kdd.[Km[dodaj]] (WyrDanA , WyrDanA) |
 Kdd[Km[dziel]] (WyrDanA , WyrDanA) |

wyrażenia słowowe

Kdd[Km[łącz]](WyrDanA, WyrDanA) |

wyrażenia listowe

Kdd[Km[twórz-li]] (WyrDanA)	
Kdd[Km[położ-na-li]] (WyrDanA, WyrDanA)	
Kdd[Km[weź-z-li]] (WyrDanA)	
Kdd[Km[usuń-z-li]] (WyrDanA)	

wyrażenia tablicowe

Kdd[Km[twórz-ta]] (WyrDanA)	
Kdd[Km[położ-ta]] (WyrDanA, WyrDanA)	
Kdd[Km[wymień-w-ta]] (WyrDanA, WyrDanA, WyrDanA)	
Kdd[Km[weź-z-ta]] (WyrDanA, WyrDanA)	

wyrażenia rekordowe

Kdd[Km[twórz-re]] (Identyfikator, WyrDanA)	
Kdd[Km[dołoż-do-re]] (Identyfikator, WyrDanA, WyrDanA)	
Kdd[Km[weź-z-re]] (WyrDanA, Identyfikator)	
Kdd[Km[usuń-z-re]] (Identyfikator, WyrDanA)	
Kdd[Km[wymień-w-re]] (WyrDanA, Identyfikator, WyrDanA)	

wyrażenia warunkowe

gdy (WyrDanA , WyrDanA , WyrDanA)

Składnię abstrakcyjną wyrażeń typologiczno-transferowych i typologicznych opisują dwa poniższe równania:

Wyrażenia transferowe

wtr : WyrTraA =

wyrażenia przetwarzające

{Kt[twórz-lic.lic.()] lic : LiczbaS}	
{Kt[twórz-sł.sło.()] sło : SłowoS}	
Kt[dodaj] (WyrTraA, WyrTraA)	
Kt[dziel] (WyrTraA, WyrTraA)	
Kt[suma] (WyrTraA)	
Kt[maks] (WyrTraA)	

wyrażenia jarzmowe

Kt[równe-lic] (WyrTraA, WyrTraA)	
Kt[mniejsze-lic] (WyrTraA, WyrTraA)	
Kt[mała-lic] (WyrTraA)	

Kt[rosnąco-lc] (WyrTraA)	
{Kt[twórz-bo.bo.()] boo : Boolowska}	
oraz-T (WyrTraA, WyrTraA)	
lub-T (WyrTraA, WyrTraA)	
nie-T (WyrTraA, WyrTraA)	
wyrażenia strukturalne	
twórz-dla-li (WyrTraA)	
twórz-dla-ta (WyrTraA)	
twórz-dla-re (Identyfikator, WyrTraA)	
wyrażenia selekcji	
Kt[weż-tr-li]	
Kt[weż-z-ta] (WyrTraA)	
Kt[weż-z-re] (Identyfikator)	
wyrażenie systemowe	
przekaż ()	

Wyrażenia typologiczne

wty : WyrTypA =

{Kdt[Ky[twórz-bo.bo.()]] boo : Boolowska}	
{Kdt[Ky[twórz-lc.lic.()]] lic : LiczbaS}	
{Kdt[Ky[twórz-sł.sło.()]] sł : SłowoS}	
stała-typ (Identyfikator)	
Kdt[Ky[twórz-li]] (WyrTypA)	
Kdt[Ky[twórz-ta]] (WyrTypA)	
Kdt[Ky[twórz-re]] (Identyfikator, WyrTypA)	
Kdt[Ky[dołoż-do-re]] (WyrTyp, Identyfikator, WyrTypA)	
Kdt[wymień-ty-tr]] (WyrTypA, WyrTraA)	

W tym miejscu warto raz jeszcze cofnąć się do rozdziału 2.12 i przypomnieć przyjęte tam konwencje notacyjne. Otóż występujące w nich metawyrażenia, np.

Kdt[Ky[twórz-li]] (WyrTypA)

należy traktować jako metanazwy funkcji składniowych, a więc w tym przypadku jest to funkcja, która każdy napis reprezentowany przez metazmienną *wty* przekształca na napis:

‘Kdt[Ky[twórz-li]] (‘ © wty © ‘)’

5.4.2 Składnia konkretna języka Lingua-A

Jak już pisałem w rozdziałach 2.14 i 4.5, historycznie *składnią konkretną* języka programowania nazywano tę składnię, która była udostępniana użytkownikowi. W naszym ujęciu składnia konkretna to dopiero przybliżenie (choć dość bliskie) przyszłej składni programisty, przy czym jest to przybliżenie o tej własności, że istnieje dla niego semantyka denotacyjna. Ostateczną składnię programisty definiujemy wprowadzając do składni konkretnej konwencje notacyjne zwane *kolokwializmami* (rozdział 4.5). Następnie określamy funkcję przekształcającą wyrażenia składni programisty w wyrażenia składni konkretnej. Tę konstrukcję opisuje Rys. 4.5-2 w rozdziale 4.5.

Niniejszy rozdział zawiera szkic składni konkretnej **Lingua-A**, który ma służyć pokazaniu jak taką składnię budujemy, ale niekoniecznie jak taka składnia ma wyglądać dla rzeczywistego języka⁶⁵. Odpowiadającą tej składni algebrę oznaczę przez AlgWyr (algebra wyrażen). Jej nośniki są zdefiniowane gramatyką równaniową (poniżej), a jej konstruktory są implicite wskazane przez równania gramatyki.

Opisane niżej modyfikacje składni abstrakcyjnej odpowiadają homomorfizmowi KO z Rys. 4.5-2, który w tym przypadku będzie sklejał jedynie wyrażenia obejmujące operator łączenia słów zarówno w kategorii wyrażen danologicznych, jak i typologiczno-transferowych. Wszędzie poza tym będzie miał cechy izomorfizmu, czyli nie będzie wprowadzał żadnych sklejeń (będzie różnowartościowy). Podstawowe zasady ogólne dotyczące wprowadzanych zmian są następujące:

1. dla napisów w składni konkretnej i kolokwialnej wprowadzam krój czcionki Courier New przy czym literowe nazwy konstruktorów nie-zeroargumentowych, zwane powszechnie *słowami kluczowymi* piszę czcionką pogrubioną **Courier New** w języku angielskim⁶⁶,
2. jako nazwy stałych boolowskich przyjmuję `true` i `false`,
3. nazwy stałych liczbowych piszę analogicznie przyjmując przecinek jako separator pomiędzy częścią całkowitą i ułamkową, np. `23,472`,
4. stałe słowowe ujmuję w cudzysłowy proste, np. `'nazwisko'`,
5. w przypadku zmiennych danologicznych i stałych typologicznych zamiast `zmienna-dan(abc)` i `stała-typ(abc)` piszę w obu przypadkach `abc`; to sklejenie nie psuje homomorfizmu (izomorfizmu), bo sklepane wyrażenia należą do różnych nośników algebry,
6. dla operatorów arytmetycznych i predykatów wprowadzam notację infiksową w miejsce prefiksowej, oraz „zwykłe” symbole operacji `+`, `/`, `<` a więc piszę np. `(x + y)` oraz `(x < y)` zamiast `dodaj(x, y)` oraz `mniejsze(x, y)` itd.; nadmiarowe nawiasy zostaną usunięte dopiero na poziomie składni kolokwialnej, ponieważ ten zabieg nie odpowiada przekształceniu homomorficznemu (rozdział 2.14),
7. jako symbol operatora składania słów przyjmuję **glue** i również dla niego stosuję notację infiksową, a dodatkowo jeszcze opuszczam nawiasy, a więc piszę np. `a glue b glue c` zamiast `(a glue b) glue c`; łączność operacji sklejanego gwarantuje,

⁶⁵ Jak już zapowiadałem wcześniej, nie jest moim celem zaproponowanie konkretnych rozwiązań dla języka **Lingua**, a jedynie zilustrowanie zasad jego budowania.

⁶⁶ Składnię konkretną i kolokwialną języków z grupy **Lingua** będę budował na gruncie języka angielskiego, co jest już powszechnie uznanym standardem. Dodatkową korzyścią takiego wyboru jest wyraźniejsze odróżnienie poziomu denotacji od składni, niż tylko przez użycie różnych krojów czcionki.

że tak zdefiniowany homomorfizm nie wyprowadza poza model denotacyjny (twierdzenie 2.13-1); więcej na ten temat w rozdziale 6.2.2 poświęconemu składce konkretnej języka **Lingua-1**,

8. dla operatorów logicznych oraz konstruktorów wprowadzam nazwy anglojęzyczne, a więc **or**, **and**, **not** oraz notację infiksową; w kontekście wyrażeń danologicznych odpowiadają one konstruktorom McCarthy’ego, a w kontekście wyrażeń transferowych — konstruktorom Kleene’ego; nie prowadzi to do nieporozumień, bo konteksty wskażą właściwe rozumienie,
9. dla wyrażenia warunkowego wprowadzam zapis infiksowy:
`if WyrDan then WyrDan else WyrDan fi,`
 i podobne konwencje przyjmuję dla wyrażeń listowych, tablicowych i rekordowych (patrz niżej),
10. przyjmuję konwencję, że wyrażenia zarówno typologiczne jak i danologiczne, dla których wybieram notację prefiksową, zamykam nawiasem **ee** co można czytać z angielska jako *end of expression*.

Nowa algebra składni jest homomorficzna z poprzednią, a odpowiadający jej homomorfizm **KO** nie skleja za dużo, co na mocy Tw. 2.13-1 gwarantuje nam istnienie jednoznacznie wyznaczonej (jedynej) semantyki denotacyjnej odwzorowującej nową składnię w naszą algebrę denotacji.

Nasza nowa gramatyka ma zapisaną niżej postać. Tym razem po nazwie kategorii składniowej nie piszę żadnego przyrostka, bo mamy do czynienia z gramatyką przeznaczoną dla użytkownika języka, który o składni abstrakcyjnej w ogóle nie będzie wiedział.

ide : Identyfikator =

`ide | ...` (dla każdego reprezentowalnego **ide**)

W tej klauzuli zastosowałem nie do końca formalną konwencję notacyjną omijającą notację teoriomnogościową użytą przy definiowaniu składni abstrakcyjnej (rozdział 5.4.1). W tym przypadku **ide** oznacza składniową reprezentację identyfikatora **ide**, a zapis:

`ide | ...` (dla każdego **ide** : Identyfikator)

oznacza zbiór:

`{ide | ide : Identyfikator}`

Analogiczną konwencją posłużę się w klauzuli definiującej składnię wyrażeń danologicznych:

Wyrażenia danologiczne

wyd : WyrDan =

stale

<code>true false</code>		
<code>lic</code>		(dla każdej <code>lic</code> : LiczbaS)
<code>sł_o</code>		(dla każdego <code>sł_o</code> : SłowoS)

zmienne

Identyfikator	(opuszczam nazwę konstruktora)
wyrażenia boolowskie	
(WyrDan and WyrDan)	
(WyrDan or WyrDan)	
(not WyrDan)	
(WyrDan < WyrDan)	
wyrażenia arytmetyczne	
(WyrDan + WyrDan)	
(WyrDan / WyrDan)	
wyrażenia słowowe	
WyrDan glue WyrDan	(opuszczenie nawiasów!)
wyrażenia listowe	
list WyrDan ee	
push WyrDan on WyrDan ee	
top (WyrDan)	
pop (WyrDan)	
wyrażenia tablicowe	
array WyrDan ee	
add-to-arr WyrDan new WyrDan ee	
change-arr WyrDan at WyrDan by WyrDan	
arr WyrDan at WyrDan ee	
wyrażenia rekordowe	
record Identyfikator of-value WyrDan ee	
add-attr Identyfikator of-value WyrDan to WyrDan ee	
rec WyrDan at Identyfikator ee	
remove-attr Identyfikator from WyrDan ee	
change-rec WyrDan at Identyfikator by WyrDan ee	
wyrażenia warunkowe	
if WyrDan then WyrDan else WyrDan fi	

Wyrażenia transferowe

wtr : WyrTra =

wyrażenia przetwarzające

lic | dla każdej lic : LiczbaS

<code>słō</code>	dla każdego <code>słō</code> : <code>SłowoS</code>
<code>(WyrTra + WyrTra)</code>	
<code>(WyrTra / WyrTra)</code>	
<code>sum (WyrTra)</code>	
<code>max (WyrTra)</code>	
wyrażenia transferowo-jarzmowe	
<code>true false</code>	
<code>(WyrTra = WyrTra)</code>	
<code>(WyrTra < WyrTra)</code>	
<code>small-number (WyrTra)</code>	
<code>increasing (WyrTra)</code>	
<code>(WyrTra and WyrTra)</code>	
<code>(WyrTra or WyrTra)</code>	
<code>(not WyrTra)</code>	
wyrażenia kwantyfikatorskie	
<code>all-list WyrTra ee</code>	
<code>all-array WyrTra ee</code>	
transfery selekcji	
<code>top</code>	
<code>array[WyrTra]</code>	
<code>record.Identyfikator</code>	
transfer przekazujący	
<code>value</code>	

Wyrażenia typologiczne

<code>wyt :WyrTyp =</code>	
<code>boolean</code>	
<code>number</code>	
<code>word</code>	
<code>Identyfikator</code>	
<code>list-type WyrTyp ee</code>	
<code>array-type WyrTyp ee</code>	
<code>record-type Identyfikator as WyrTyp ee</code>	
<code>expand-record-type WyrTyp at Identyfikator by WyrTyp ee</code>	

replace-transfer-in WyrTyp by WyrTra ee

Jak już wskazywałem, jedynym „miejszem sklejenia” składni abstrakcyjnej przy przejściu do składni konkretnej jest opuszczenie nawiasów związanych z operacją **glue**, która odpowiada operacji łącznej konkatenacji słów. Pozornie mogłoby się wydawać, że sklejenie ma miejsce również wtedy, gdy wyrażenie danologiczne i transferowe są „sklejane” w jedno konkretne, jak np.

```
Kdd[Km[twórz-bo.pp]]
```

```
Kt[twórz-bo.pp]
```

zostają sklejone w konkretne

```
true
```

jednak, że nie zachodzi tu przypadek sklejącego homomorfizmu, bo każde z tych wyrażeń należy do innego nośnika algebry składni.

5.4.3 Składnia kolokwialna języka Lingua-A

Definiowanie składni kolokwialnej to bardzo ważny etap budowania języka programowania, gdyż wtedy właśnie język może nabrać cech przyjaznych dla użytkownika. Na tym etapie uwalniamy się od algebraicznych rygorów składni konkretnej nie tracąc matematycznej precyzji, a zyskując na przejrzystości.

Wobec składni kolokwialnej przyjmuję zasadę, że obejmuje ona całą składnię konkretną plus kolokwializmy, co oznacza, że używanie kolokwializmów jest opcjonalne. Z kolei na poziomie gramatyki języka oznacza to nową regułę gramatyczną dla każdego kolokwializmu, co oczywiście powoduje, że algebra składni kolokwialnej nie jest podobna do algebry składni konkretnej, bo ma więcej konstruktorów.

Poniżej podaję przykładowe kolokwializmy związane z operacjami na danych prostych, tablicowych i rekordowych. Nie twierdzę, że są one najlepsze z możliwych, gdyż chodzi mi raczej o ilustrację metody, a nie o zbudowanie praktycznego języka. Wszystkie kolokwializmy zdefiniuję w sposób nieformalny posługując się przykładami, co powinno jednak wystarczyć do zdefiniowania zarówno reguł gramatycznych jak i transformacji przywracających.

5.4.3.1 Reguły uniwersalne

Te zasady odnoszą się do wszystkich rodzajów wyrażeń:

1. dopuszczam spacje i nawroty karetki, które przez transformację przywracającą będą usuwane,
2. żadne ze słów kluczowych `true`, `false`, `if`, `then`,... nie może być użyte jako identyfikator; w tym przypadku transformacja przywracająca nie zmieni niczego w programie, ale sygnalizuje błąd składniowy; w tradycyjnych parserach ta analiza jest włączona do analizy leksykalnej.

5.4.3.2 Danologiczne wyrażenia boolowskie

Dla operacji boolowskich wprowadzam szkolne zasady opuszczania nawiasów, co oznacza, że transformacja przywracająca będzie je dopisywać zgodnie z regułą priorytetu koniunkcji nad alternatywą. Na przykład,

- zamiast napisać `(x or (y or z))` napiszemy `x or y or z`,

- zamiast napisać $(x \text{ or } (y \text{ and } z))$ napiszemy $x \text{ or } y \text{ and } z$

W pierwszym przypadku transformacja przywracająca może wstawić nawiasy w dowolny sposób, jako konsekwencja łączności alternatywy, w drugim — musi uwzględnić priorytety pomiędzy działaniami.

5.4.3.3 Danologiczne wyrażenia liczbowych

Przypadek wyrażeń liczbowych jest nieco bardziej złożony, gdyż w sytuacjach rzeczywistych, gdy mamy do dyspozycji cztery działania arytmetyczne (a nie dwie, jak w naszym uproszczonym przykładzie), to zarówno dodawanie, jak i mnożenie nie są łączne. Jest to wynikiem zjawiska powstawania nadmiaru. Na przykład, gdyby największą liczbą dopuszczalną w arytmetyce komputera było 10, to

$$((-4 + 9) + 2) = 7 \quad \text{jednakże}$$

$$(-4 + (9 + 2)) = \text{'nadmiar'}$$

Zwykle przyjmuje się więc, że wyrażenia beznawiasowe wykonuje się „od lewej do prawej”, co oznacza, że funkcja przywracająca wyrażenie

$$x + y + z + x * y$$

zamieni na

$$((x + y) + z) + (x * z).$$

5.4.3.4 Danologiczne wyrażenia tablicowe

Dla tablicowych wyrażeń danologicznych wprowadzam cztery kolokwializmy. Pierwszy dotyczy konstruktora tworzenia tablicy. Na przykład wyrażenie kolokwialne:

```
array [x, x+y, 3*y]
```

rozwija się do wyrażenia konkretnego:

```
add-to-arr (dodaj do tablicy wartość 3*y)
```

```
add-to-arr (dodaj do tablicy wartość x+y)
```

```
array x ee (utwórz tablicę jednoelementową z wartością x)
```

```
new x+y ee
```

```
new 3*y ee
```

Oczywiście w miejscu występujących tu prostych wyrażeń arytmetycznych mogą stać dowolne wyrażenia. Jeżeli wyniki-pomiarów jest zmienną tablicową, to wyrażenie kolokwialne:

```
wyniki-pomiarów.[x+1]
```

rozwija się do wyrażenia konkretnego

```
arr wyniki-pomiarów at x+1 ee
```

natomiast wyrażenie kolokwialne

```
wyniki-pomiarów.[x+1].[y-1]
```

rozwija się do konkretnego:

```
arr arr wyniki-pomiarów at x+1 ee at y-1 ee
```

W przypadku dodawania do tablic nowych elementów możemy napisać analogicznie jak przy tworzeniu tablic:

add-to-arr wyniki-pomiarów **new** [x, x + y, 3*y] **ee**

a w przypadku modyfikacji (wprowadzam nowy symbol „<=”):

change-arr wyniki-pomiarów **by**

```
s    <= x,
s+1  <= x+y,
3*p  <= z-1
```

ee

co przekłada się na:

change-arr

change-arr

change-arr wyniki-pomiarów **at** s **by** x **ee**

at s+1 **by** x+y **ee**

at 3*p **by** z-1 **ee**

5.4.3.5 Danologiczne wyrażenia rekordowe

Przykładowe kolokwializmy dla rekordów mogą być podobne do tablicowych. Na przykład możemy przyjąć, że wyrażenie kolokwialne

record

```
imię          <= 'Jan',
nazwisko      <= 'Kowal',
rok-urodzenia <= 1968,
lata-nagród   <= lata-nagród-kowal
```

ee

odpowiada konkretnemu:

add-atr lata-nagród **of-value** lata-nagród-kowal **to**

add-atr rok-urodzenia **of-value** 1968 **to**

add-atr nazwisko **of-value** 'Kowal' **to**

set-record imię **of-value** 'Jan'

ee

ee

ee

ee

a wyrażenie kolokwialne

pracownik.(nazwisko)

odpowiada konkretnemu

rec pracownik **at** nazwisko **ee**

Zauważmy, że mimo podobieństwa wyrażeń pobierania z rekordu i z tablicy, nie pojawia się tu niejednoznaczność, bo indeksy tablicowe są ujmowane w nawiasu kwadratowe, a atrybuty rekordu w nawiasy zwykłe. Zatem gdyby `pracownik` była zmienną tablicową, a `nazwisko` zmienną liczbową, to wyrażenie pobierania miałoby postać:

```
pracownik.[nazwisko]
```

5.4.3.6 Transferowe wyrażenia tablicowe

Wprowadzam szkolne zasady priorytetów operacji i opuszczania nawiasów. Na przykład w miejsce

```
(2+value) < 10
```

napizę

```
2+value < 10
```

W miejsce

```
get-from-array x+1 ee
```

napizę

```
array.[x+1]
```

Przypominam, że w tym przypadku `array` nie jest zmienną tablicową — jak np. w wyrażeniu danologicznych `pomiary.[x+1]` — ale słowem kluczowym, które przypomina, że kompozyt wejściowy tego tranzytu powinien nieść tablicę, a wyrażenie transferowe pobierze z tej tablicy element o indeksie będącym wartością wyrażenia `x+1`.

5.4.3.7 Typologiczne wyrażenia rekordowe

W przypadku rekordowych wyrażeń typologicznych wprowadzam kolokwializmy analogiczne jak dla wyrażeń danologicznych. Na przykład

```
record-type
```

```
    imię          as word,
```

```
    nazwisko      as word,
```

```
    rok-urodzenia as number,
```

```
    lata-nagród   as array-of number ee
```

```
ee
```

rozwija się do wyrażenia konkretnego zapisanego za pomocą `record-of` oraz `expand-record`.

5.4.3.8 Transferowe wyrażenia rekordowe

W tym przypadku podobnie jak dla tablic piszemy:

```
record.nazwisko
```

co oznacza

```
get-from-record nazwisko ee
```

Zatem w pierwszym z tych wyrażeń `record` jest słowem kluczowym podobnie jak `array` w przypadku tablic.

5.4.3.9 Wyrażenia typologiczne

W większości języków programowania jarzma nie występują w definicjach typów, co w składni konkretnej zapisalibyśmy jako:

```
set-type WyrTyp with true ee
```

a więc na przykład:

```
set-type array-of number ee with true ee
```

W takim przypadku kolokwialny odpowiednik tego wyrażenia będzie

```
set-type  
  array-of number ee  
ee
```

Zasada ogólna jest więc taka, że jeżeli jarzmem miałyby być stała `true` to opuszczamy całą frazę „**with** `true`”.

W przypadku typów rekordowych wprowadzam kolokwializm pozwalający opisać jarzma wraz z korpusem jednym wyrażeniem. Na przykład piszemy

```
record-type  
  imię          as word,  
  nazwisko      as word,  
  rok-urodzenia as number with small-number,  
  lata-nagród   as array-of number with small-number ee  
ee
```

co oznacza

```
type  
record-of  
  imię          as word,  
  nazwisko      as word,  
  rok-urodzenia as number,  
  lata-nagród   as array-of number ee  
ee  
with  
  small-number(record.rok-urodzenia) and  
  all-of-array record.lata-nagród with small-number(value) ee  
ee
```

Jarzmo tego typu rekordowego jest spełnione, jeżeli są spełnione łącznie trzy warunki:

1. kompozyt niesie rekord z co najmniej dwoma atrybutami `rok-urodzenia` oraz `lata-nagród`.
2. z atrybutem `rok-urodzenia` jest związana liczba mała,

3. z atrybutem `lata-nagród` jest związana tablica, której wszystkie elementy są liczbami małymi.

Pozostałe informacje o typie rekordu są zawarte w wyrażeniu typologicznym.

5.5 Zadania budowniczego języka

Budowniczy języka programowania ma do zrealizowania trzy kreatywne zadania wymagające podejmowania decyzji co do funkcjonalności i kształtu swojego przyszłego dzieła:

1. zbudowanie algebry denotacji `AlgDen`, która w sposób jednoznaczny wyznacza algebrę składni abstrakcyjnej `AlgSkłAbs`,
2. zbudowanie homomorficznej do niej algebry składni konkretnej `AlgSkł`, która nie będzie bardziej wieloznaczna niż `AlgDen`,
3. zdefiniowanie składni kolokwialnej i związanej z nią transformacji przywracającej.

Jeżeli składnia konkretna została prawidłowo zbudowana, to semantyka języka, czyli homomorfizm algebr:

$$\text{Sem} : \text{AlgSkł} \mapsto \text{AlgDen}$$

istnieje i jest jednoznacznie wyznaczony.

Stworzenie implementacji języka programowania polega na napisaniu procedur, które każdy zapisany w składni kolokwialnej ciąg znaków, oznaczmy go przez `program-kol`, poddają trójstopniowej obróbce odpowiadającej transformacji przywracającej `Tp` oraz dwóm homomorfizmom `Ko` i `Sa` z Rys. 4.5-2

Krok pierwszy dokonuje stosunkowo prostej transformacji ze składni kolokwialnej `program-kol` do konkretnej `program-kon`. Oczywiście już w trakcie tej transformacji może pojawić się sygnał błędu.

Krok drugi jest realizowany przez *analizator składniowy* języka, zwany też *parserem*, który odtwarza przeciwobraz napisu `program-kon` w algebrze składni abstrakcyjnej. Ten przeciwobraz — oznaczmy go jako `program-abs` — jest niczym innym jak liniowym zapisem tzw. *drzewa parsingu* dla programu `program-kon`. Na gruncie naszego modelu spełnia on równanie:

$$\text{Ko}[\text{program-abs}] = \text{program-kon}$$

Jeżeli składnia konkretna nie jest jednoznaczna, czyli `Ko` jest homomorfizmem sklejającym, to parser jest zbudowany tak, aby wybrał jeden z przeciwobrazów napisu `program-kon`. Ponieważ składnia konkretna skleja nie więcej niż algebra denotacji, ten wybór nie będzie miał wpływu na denotację naszego programu.

Gdy dla danego programu próba zbudowania drzewa parsingu się nie powiedzie, użytkownik jest informowany, że dany program zawiera błędy składniowe, co oznacza, że nie należy do języka opisywanego gramatyką składni konkretnej. Zwykle jest też wtedy generowany komunikat pozwalający zlokalizować błąd.

Jeżeli krok drugi nie zakończy się komunikatem błędu, to krok trzeci polega na wykonaniu programu. Na gruncie naszego modelu odpowiada to odtworzeniu denotacji programu `den.program` spełniającej równanie:

$$\text{den-program} = \text{Sa}[\text{program-abs}]$$

Implementacyjnie odpowiada to wykonaniu programu przez *interpreter* lub wygenerowaniu kodu w języku maszyny przez *kompilator*. W dalszym ciągu będę głównie mówił o interpreterach jako intuicyjnie bliższych semantyce denotacyjnej, jednakże wszystkie obserwacje poczynione dla interpreterów będą się odnosiły również do kompilatorów.

Twórca implementacji języka ma więc za zadanie stworzenie trzech podstawowych narzędzi komputerowych:

1. analizatora transformującego składnię kolokwialną na konkretną,
2. parsera składni konkretnej na abstrakcyjną,
3. interpretera (bądź kompilatora) składni abstrakcyjnej.

Zadania drugie i trzecie powinny dać się zrealizować algorytmicznie wychodząc z gramatyki konkretnej języka⁶⁷ oraz z definicji konstruktorów denotacji.

Język powinien być zbudowany w ten sposób, aby jak najwięcej błędów dawało się wykryć na poziomie analizy składni, gdyż jest ona znacznie szybsza od fazy wykonywania programu. Staramy się więc jak najwięcej cech języka zapisać na tym poziomie, co oznacza stworzenie maksymalnie wielu nośników w algebrze denotacji. Na przykład, dobrze zbudowana gramatyka składni konkretnej języka powinna pozwolić na wykrycie na poziomie parsingu błędu składniowego w instrukcji

```
if y > 0 then y+1 else list-type number ee fi
```

gdzie po **else** stoi wyrażenie typologiczne, a nie danologiczne. Natomiast na poziomie składni nie jesteśmy w stanie zagwarantować, o czym już pisałem, by zmienne użyte w wyrażeniach liczbowych miały typ liczbowy, a w wyrażeniach boolowskich — boolowski. Błędy tego rodzaju są więc wykrywane dopiero w czasie wykonywania programu⁶⁸.

5.6 Dwie formuły podręcznika języka programowania

Denotacyjny model języka programowania stanowi punkt wyjścia nie tylko do stworzenia implementacji, ale także do napisania podręcznika użytkownika. Ponieważ tak pisany podręcznik jeszcze nigdy nie powstał (mam nadzieję, że powstanie dla **Lingua**) nie została jeszcze wypracowana żadna praktyka w tym obszarze. Wydaje się jednak oczywiste, że powinien on opisywać język w trzech krokach i w niżej określonej kolejności:

1. składnia konkretna opisana gramatyką równaniową i ilustrowana przykładami,
2. odpowiadająca jej składnia kolokwialna ilustrowana przykładami transformacji przywracających (np. jak w rozdziale 5.4.3),
3. semantyka składni konkretnej, a więc przypisanie programom konkretnym ich denotacji bez odwoływania się do składni abstrakcyjnej.

Opisując semantykę języka trzeba wybrać pomiędzy dwiema postaciami jej definicji o czym pisałem już w rozdziale 5.3.3:

⁶⁷ Metody generowania parserów z gramatyk bezkontekstowych znane są już od wczesnych lat 1960.

⁶⁸ W rzeczywistości błędy związane z typowaniem zmiennych mogą być wykrywane na poziomie tzw. *semantyki statycznej*, która ogranicza się do wyliczania typów zmiennych (w naszym przypadku korpusów) bez wyliczania wartości. Jest to jednak nadal semantyczny poziom opisu języka, a nie poziom składniowy. Takie rozwiązanie stosowano przy budowaniu semantyki języka Ada [12] za pomocą metodologii VDM (Vienna Development Method) [10].

- A. może ona odwoływać się do wcześniej zdefiniowanych konstruktorów algebry denotacji; taką postać definicji nazwę *algebraiczną*,
- B. może przytaczać definicje konstruktorów explicite; tę wersję definicji nazwę *wyraźną*.

Którą z tych postaci wybierzemy, zależy od adresata definicji.

Sądzę, że dla twórców implementacji wygodniejsza jest postać algebraiczna. Definicje konstruktorów denotacji można bowiem zapisać jako układ wzajemnie wywołujących się (rekurencyjnie) procedur, a równania definiujące semantykę jako również rekurencyjny układ procedur wywołujących procedury konstruktorów.

Z kolei dla użytkownika języka wygodniejszą wydaje się definicja semantyki drugiego rodzaju, która dla każdej konstrukcji składniowej podaje jej pełen semantyczny opis nie wymagający sięgania do algebr denotacji.

5.7 Szkic semantyki Lingua-A

Przypomnijmy że AlgWyr i odpowiednio przez AlgDenWyr zostały oznaczone algebry składni konkretnej i denotacji **Lingua-A**. Ponieważ pierwsza z tych algebr nie jest bardziej wieloznaczna od drugiej, istnieje i jest jednoznacznie wyznaczony homomorfizm:

$$Sk : Skk.Lingua-A \mapsto Den.Lingua-A$$

o pięciu składowych:

$$Sid : Identyfikator \mapsto Identyfikator$$

$$Swd : WyrDan \mapsto DenWyrDan$$

$$Swtr : WyrTra \mapsto DenWyrTra$$

$$Swt : WyrTyp \mapsto DenWyrTyp$$

Poniżej opiszę te składowe ograniczając się do przykładów ich równań definicyjnych, ale za to pokażę każde w dwóch wersjach: algebraicznej i wyraźnej. Przyjmę konwencję, że czcionką Courier składam nie tylko konkretne elementy składni języka, ale też przebiegające je metazmienne. Przypominam, że „Km[o_{pe}]” to konstruktor kompozytów odpowiadający operacji na danych o_{pe} (rozdział 5.2.3), natomiast Kdd[Km[o_{pe}]] to odpowiadający mu konstruktor denotacji wyrażeń danologicznych.

Identyfikatory

$$Sid : Identyfikator \mapsto Identyfikator$$

$$Sid.[ide] = \text{twórz-id.ide.()} \quad \text{dla każdego ide} \quad \text{(postać algebraiczna)}$$

$$Sid.[ide] = ide \quad \text{dla każdego ide} \quad \text{(postać wyraźna)}$$

Wyrażenia danologiczne

$$Swd : WyrDan \mapsto DenWyrDan \quad \text{czyli}$$

$$Swd : WyrDan \mapsto Stan \rightarrow \text{KompozytB}$$

$\text{Swd.}[\text{true}] = \text{Kdd}[\text{twórz-bo-km.pp}]().()$ (postać algebraiczna)

$\text{Swd.}[\text{true}].\text{sta} =$ (postać wyraźna)

$\text{jest-błąd.sta} \rightarrow \text{błąd.sta}$

prawda $\rightarrow (\text{pp}, (\text{'boolowska'}))$

$\text{Swd.}[\text{ide}] = \text{zmienna-dan.}(\text{Sid.}[\text{ide}])$ dla $\text{ide} : \text{Identyfikator}$

$\text{Swd.}[\text{ide}].\text{sta} =$

$\text{jest-błąd.sta} \rightarrow \text{błąd.sta}$

niech

$(\text{śro}, (\text{wrt}, \text{'OK'})) = \text{sta}$

$\text{wrt.ide} = ? \rightarrow \text{'zmienna-niezadeklarowana'}$

niech

$((\text{dan}, \text{kor}), \text{jar}) = \text{wrt.ide}$

$\text{dan} = \Omega \rightarrow \text{'zmienna niezainicjalizowana'}$

prawda $\rightarrow (\text{dan}, \text{kor})$

$\text{Swd.}[(\text{wyd-1} + \text{wyd-2})] = \text{Kdd}[\text{dodaj-km}].(\text{Swd.}[\text{wyd-1}], \text{Swd.}[\text{wyd-2}])$

$\text{Swd.}[(\text{wyd-1} + \text{wyd-2})].\text{sta} =$

$\text{jest-błąd.sta} \rightarrow \text{błąd.sta}$

$\text{Swd.}[\text{wyd-i}].\text{sta} = ? \rightarrow ?$ dla $i = 1, 2$

niech

$(\text{dan-i}, \text{kor-i}) = \text{Swd.}[\text{wyd-i}].\text{sta}$ dla $i = 1, 2$

$\text{kor-i} \neq (\text{'liczba'}) \rightarrow \text{'oczekiwana-liczba'}$ dla $i = 1, 2$

niech

$\text{lic} = \text{zaokrąglij.}(\text{dodaj.}(\text{dan-1}, \text{dan-2}))^{69}$

$\text{kom} = (\text{lic}, (\text{'liczba'}))$

$\text{nadmiarowy.kom} \rightarrow \text{'przekroczenie-rozmiaru'}$

prawda $\rightarrow (\text{lic}, (\text{'liczba'}))$

Wyrażenia transferowe

Przypominam, że denotacje wyrażeń transferowych to transfery, stąd więc w poniższych klauzulach pojawia się metazmienna kom .

$\text{Swtr} : \text{WyrTra} \mapsto \text{Transfer}$

⁶⁹ W tym miejscu korzystamy z faktu, że kompozyty są dobrze ustrukturyzowane, a więc skoro $\text{kor-i} = (\text{'liczba'})$ dla $i = 1, 2$, to $\text{dan-i} : \text{Liczba}$ dla $i = 1, 2$.

$\text{Swtr} : \text{WyrTra} \mapsto \text{KompozytB} \mapsto \text{KompozytB}$

$\text{Swtr}[\text{true}] = \text{twórz-tr-bo.pp.}()$

$\text{Swtr}[\text{true}].\text{kom} =$

$\text{kom} : \text{Błąd} \rightarrow \text{kom}$

prawda $\rightarrow (\text{pp}, ('boolowska'))$

$\text{Swtr}[\text{value}] = \text{przekaż.}()$

$\text{Swtr}[\text{value}].\text{kom} = \text{kom}$

$\text{Swtr}[\text{all-list wtr-l with wtr-b ee}] =$

$\text{wszystkie-na-li.}(\text{Swtr}[\text{wtr-l}], \text{Swtr}[\text{wtr-b}])$

$\text{Swtr}[\text{all-list wtr-l with wtr-b ee}].\text{kom} =$

$\text{kom} : \text{Błąd} \rightarrow \text{kom}$

niech

$\text{kom-l} = \text{Swtr}[\text{wtr-l}].\text{kom}$

$\text{kom-l} : \text{Błąd} \rightarrow \text{kom-l}$

$\text{rodzaj.kom-l} \neq 'L' \rightarrow \text{'oczekiwana-lista'}$

niech

$(\text{dan-1}, \dots, \text{dan-n}) = \text{dana.kom-l}$

$('L', \text{kor}) = \text{korpus.kom-l}$

$\text{kom-i} = \text{Swtr}[\text{wtr-b}].\text{sta.}(\text{dan-i}, \text{kor}) \quad \text{dla } i = 1;n$

$\text{kom-i} : \text{Błąd} \rightarrow \text{kom-i}$

$(\forall i = 1;n) \text{kom-i} = (\text{pp}, ('boolowska')) \rightarrow (\text{pp}, ("boolowska"))$

prawda $\rightarrow (\text{ff}, ("boolowska"))$

Wyrażenia typologiczne

Denotacje wyrażeń typologicznych sięgają do typów zapisanych w środowisku typów.

$\text{Swty} : \text{WyrTyp} \mapsto \text{DenWyrTyp}$

$\text{Swty} : \text{WyrTyp} \mapsto \text{Stan} \mapsto \text{Typ} \mid \text{Błąd}$

$\text{Swty}[\text{ide}] = \text{stała-typ.ide}$

$\text{Swty}[\text{ide}].\text{sta}$

$\text{jest-błąd.sta} \rightarrow \text{błąd.sta}$

niech

$((\acute{s}rt, \acute{s}rp), skl) = sta$

$\acute{s}rt.ide = ? \rightarrow \text{'stała-typologiczna-niezdefiniowana'}$

prawda $\rightarrow \acute{s}rt.ide$

Zadanie napisania kolejnych definicji pozostawiam czytelnikowi.

6 Lingua-1 — język imperatywny bez procedur

Poczynając od niniejszego rozdziału będę budował kolejne języki z grupy **Lingua** przez rozbudowywanie każdego z nich o nowe mechanizmy. **Lingua-1** powstanie z **Lingua-A** przez rozbudowanie go o podstawowe konstrukcje imperatywne obejmujące instrukcje, deklaracje zmiennych i definicje typów. Procedurami zajmę się w rozdziale 7.

6.1 Denotacje

6.1.1 Dziedziny denotacyjne

Dziedziny denotacji **Lingua-1** odpowiadają dziewięciu jego przyszłym kategoriom składniowym:

1. identyfikatory,
2. wyrażenia danologiczne,
3. wyrażenia transferowe,
4. wyrażenia typologiczne,
5. deklaracje zmiennych danologicznych,
6. definicje stałych typologicznych
7. instrukcje,
8. preambuły będące sekwencjami definicji stałych typologicznych i zmiennych danologicznych,
9. programy będące parami składającymi się z preambuły i instrukcji.

Zatem nośniki algebry denotacji naszego języka są następujące:

ide	: Identyfikator	(6.1.1-1)
dwd	: DenWyrDan	= Stan $\rightarrow$ KompozytB
tra	: DenWyrTra	= Transfer
dwt	: DenWyrTyp	= Stan $\mapsto$ TypB
ddz	: DenDekZmi ⁷⁰	= Stan $\mapsto$ Stan (den. deklaracji zmiennych danologicznych)
ddt	: DenDefTyp	= Stan $\mapsto$ Stan (den. definicji stałych typologicznych)
din	: DenIns	= Stan $\rightarrow$ Stan (denotacje instrukcji)
dep	: DenPre	= Stan $\rightarrow$ Stan (denotacje preambuły)

⁷⁰ Używam pojęcia deklaracji zmiennej, a nie po prostu deklaracji, gdyż w dalszych wersjach **Lingua** wprowadzę też deklaracje procedur i funkcji.

$\text{dpr} : \text{DenPro} = \text{Stan} \rightarrow \text{Stan}$ (denotacje programów)

Zwróćmy uwagę, o czym pisałem już wcześniej, że denotacje wyrażeń danologicznych są funkcjami częściowymi. Co prawda w **Lingua-1** osiągalne denotacje wyrażeń będą funkcjami całkowitymi, jednak w **Lingua-2**, gdzie wprowadzę procedury funkcyjne, denotacje wyrażeń obejmą również funkcje częściowe. Co do instrukcji i programów, to z częściowością ich denotacji spotkamy się już teraz, gdyż wobec obecności konstruktora **while**, o którym nieco dalej, wykonywanie instrukcji, a więc też i programów, może się nigdy się nie zakończyć.

Pierwsze cztery dziedziny obejmujące denotacje aplikatywne wraz z ich konstruktorami zostały omówione w rozdziale 5. Pozostałe — obejmują różne rodzaje *denotacji imperatywnych* i są omawiane poniżej.

6.1.2 Deklaracje zmiennych danologicznych

Jak już wiemy z rozdziału 5.3.2 *zmienne danologiczne*, w skrócie *zmienne*, to identyfikatory, którym w wartościowaniach przypisano wartości lub pseudowartości. Deklaracja zmiennej wiąże jej identyfikator z typem pozostawiając (na razie) jej daną nieokreśloną. Wartości są przypisywane zmiennym przez instrukcje przypisania (rozdział 6.1.4).

$\text{deklaruj-zmi-dan} : \text{Identyfikator} \times \text{DenWyrTyp} \mapsto \text{DenDekZmi}$

$\text{deklaruj-zmi-dan}(\text{ide}, \text{dwt}).\text{sta} =$

$\text{jest-błąd}.\text{sta} \rightarrow \text{sta}$

$\text{zadeklarowany}.\text{ide}.\text{sta} \rightarrow \text{sta} \blacktriangleleft \text{'identyfikator-zajęty'}$

niech

$(\text{śro}, (\text{wrt}, \text{'OK'})) = \text{sta}$

$\text{typ} = \text{dwt}.\text{sta}$

$\text{typ} : \text{Błąd} \rightarrow \text{sta} \blacktriangleleft \text{typ}$

prawda $\rightarrow (\text{śro}, (\text{wrt}[\text{ide}/(\Omega, \text{typ})], \text{'OK'}))$

Jeżeli stan niesie komunikat błędu, to deklaracja go nie zmienia (transparentność względem błędów).

W przeciwnym przypadku, jeżeli w aktualnym stanie identyfikator *ide* został wcześniej zadeklarowany, to jest generowany błąd, który zostaje przekazany do stanu. Jak stąd w szczególności wynika, żaden identyfikator nie może być zadeklarowany dwukrotnie.

Jeżeli wyrażenie typologiczne generuje błąd, to ten błąd jest przekazywany do stanu. W przeciwnym przypadku, wartościowanie jest modyfikowane przez przypisanie identyfikatorowi *ide* pseudowartości (Ω, typ) . Deklaracje zmieniają więc jedynie wartościowania i ewentualnie generują komunikat błędu. Są to jedyne denotacje, które wprowadzają do stanów pseudowartości. O identyfikatorze, któremu przypisano wartość lub pseudowartość (dan, typ) mówimy, że *jest typu typ*.

Deklaracje zmiennych danologicznych mogą być składane sekwencyjnie. Służy do tego konstruktor:

$\text{sekwencja-dez} : \text{DenDekZmi} \times \text{DenDekZmi} \mapsto \text{DenDekZmi}$

$\text{sekwencja-dez}(\text{dez-1}, \text{dez-2}) = \text{dez-1} \bullet \text{dez-2}$

6.1.3 Definicje stałych typologicznych

Stale typologiczne to identyfikatory, którym w środowisku typów przypisano typy. Nazywam je stałymi, a nie zmiennymi, gdyż raz przypisany identyfikatorowi typ pozostaje niezmieniony przez cały czas wykonywania programu.

Jedyny konstruktor denotacji definicji stałej typologicznej, jakim będziemy się posługiwać w **Lingua**, jest następujący:

definiuj-sta-typ : Identyfikator $\times$ DenWyrTyp $\mapsto$ DenDefTyp

definiuj-sta-typ.(ide, dwt).sta =

jest-błąd.sta $\rightarrow$ sta

zadeklarowany.ide.sta $\rightarrow$ sta $\blacktriangleleft$ 'identyfikator-zajęty'

niech

typ = dwt.sta

((śrt, śrp), skł) = sta

typ : Błąd $\rightarrow$ sta $\blacktriangleleft$ typ

prawda $\rightarrow$ ((śrt[ide/typ], śrp), skł)

Jak widzimy, definicje typów zmieniają jedynie środowiska typów i ewentualnie komunikat błędu.

Podobnie jak denotacje zmiennych, tak i definicje stałych typologicznych, mogą być składowane sekwencyjnie. Służy do tego konstruktor:

sekwencja-def : DenDefTyp $\times$ DenDefTyp $\mapsto$ DenDefTyp

sekwencja-def.(ddt-1, ddt-2) = ddt-1 $\bullet$ ddt-2

6.1.4 Instrukcja przypisania

Dla zdefiniowania instrukcji przypisania wprowadzam *relację koherentności* pomiędzy korpusami. Powiem, że *korpus kor-1 jest koherentny z korpusem kor-2* co zapisuję formułą

kor-1 koherentny kor-2

gdym:

1. kor-1 = kor-2 lub
2. oba korpusy są rekordowe i zbiór atrybutów któregośkolwiek z nich jest podzbiorem zbioru atrybutów drugiego, a na wspólnym zbiorze atrybutów się pokrywają.

Innymi słowy, dwa korpusy są koherentne, jeżeli są identyczne, lub jeżeli oba są rekordowe, przy czym jeden z nich powstaje z drugiego przez dodanie lub usunięcie jakichś atrybutów. (rozdział 5.2.2). Jak łatwo sprawdzić, relacja koherencji jest zwrotna i symetryczna, ale nie przechodnia.

O denotacji imperatywnej *dim* powiemy, że jest *konserwatywna*, jeżeli spełnia dwa warunki:

1. jest transparentna względem stanów niosących komunikat błędu, tj. jeżeli *sta* niesie taki komunikat, to *dim.sta* jest określone i niesie ten sam błąd; zauważmy, że *sta* i *dim.sta* mogą się od siebie różnić (będzie tak w przypadku obsługi wyjątków opisanej w rozdziale 6.1.8),

2. jeżeli nie generuje sygnału błędu, to korpusy wszystkich zmiennych danologicznych zadeklarowanych w stanie wejściowym są koherentne z ich korpusami w stanie wyjściowym; w szczególności oznacza to, że przypisane zmiennym korpusy nierekordowe nie ulegają zmianie.

Zauważmy, że denotacje deklaracji zmiennej i definicji typu są konserwatywne. W przyszłości zadbam o to, aby wszystkie denotacje imperatywne osiągalne w naszej algebrze były konserwatywne. Ponadto wszystkie za wyjątkiem obsługi błędów nie będą zmieniały stanu, jeżeli nie sie on błąd.

O konstruktorze denotacji imperatywnych, który przyjmuje wśród swoich argumentów denotacje imperatywne powiem, że jest *rzetelny*, jeżeli przekształca denotacje konserwatywne w konserwatywne. Zadbam też o to, aby wszystkie tego rodzaju konstruktory denotacji imperatywnych były rzetelne.

Teraz możemy zdefiniować konstruktor odpowiadający instrukcji przypisania, który z identyfikatora i denotacji wyrażenia danologicznego tworzy denotację imperatywną konserwatywną:

przypisz : Identyfikator $\times$ DenWyrDan $\mapsto$ DenIns

przypisz.(ide, dwd).sta =

jest-błąd.sta $\rightarrow$ sta

niech

$((\acute{s}rt, \acute{s}rp), (wrt, 'OK')) = sta$

wrt.ide = ? $\rightarrow$ sta $\blacktriangleleft$ 'niezadeklarowany-identyfikator'

dwd.sta = ? $\rightarrow$? (niekończące się wyliczenie)

dwd.sta : Błąd $\rightarrow$ sta $\blacktriangleleft$ dwd.sta

niech

$((dan-d, kor-d), tra) = wrt.ide$ (d – dawne)

$(dan-n, kor-n) = dwd.sta$ (n – nowe)

kom = tra.(dan-n, kor-n)

kom : Błąd $\rightarrow$ sta $\blacktriangleleft$ com

nie kor-n koherentny kor-d $\rightarrow$ sta $\blacktriangleleft$ 'brak-koherencji'

not com : KompozytBoo $\rightarrow$ sta $\blacktriangleleft$ 'oczekiwane-jarzmo'

kom $\neq$ (pp, ('boolowska')) $\rightarrow$ sta $\blacktriangleleft$ 'niespełnione-jarzmo'

niech

war-n = ((dan-n, kor-n), tra)

prawda $\rightarrow ((\acute{s}rt, \acute{s}rp), wrt[ide/war-n], 'OK')$

Aby instrukcja przypisania mogła zostać wykonana, zmienna, której jest przypisywana nowa wartość, musi być uprzednio zadeklarowana. Instrukcja przypisania może zmieniać typ zmiennej, jednak wyłącznie w ten sposób, by jarzmo tra nie uległo zmianie, a nowy korpus był koherentny z poprzednim. Oczywiście nowy kompozyt musi spełniać zastane jarzmo.

Zauważmy, że w **Lingua-1** typ zmiennej rekordowej może ulegać zmianie w trakcie wykonywania programu, podczas gdy typy zmiennych nie-rekordowych nie ulegają zmianie. Jak zobaczymy w rozdziale 12.7.6.11 w **Lingua-SQL** podobną cechę będą miały zmienne tabelowe.

Pewnego komentarza wymaga sytuacja związana ze zmiennymi o typie rekordowym. Otóż przy budowaniu rekordów mamy w zasadzie dwie strategie. Pierwsza polega na zadeklarowaniu zmiennej rekordowej o jednym atrybucie, a następnie w kolejnych przypisaniach dodawaniu nowych atrybutów. Antycypując przyszłą składnie kolokwialną można to zilustrować następującym przykładem:

```
set typ-pracownik as
  record-type
    imię of type word
  ee
tes ;
let pracownik as typ-pracownik tel ;
pracownik := record imię <= 'Jan' ee ;
pracownik := add-atr nazwisko <= 'Kowal' to pracownik ee ;
pracownik := add-atr rok-urodzenia <= 1968 to pracownik ee
```

Zauważmy, że 'Kowal' to wyrażenie konkretne, któremu odpowiada wyrażenie abstrakcyjne

$\text{Kdd}[\text{Km}[\text{twórz-sł.}'\text{Kowal}'.()]]$

Denotacja tego wyrażenia tworzy następujący kompozyt (zakładając, że stan nie niesie błędu):

$\text{Kdd}[\text{Km}[\text{twórz-sł.}'\text{Kowal}'.()]].\text{sta} =$
 $(\text{twórz-sł.}'\text{Kowal}'.(), \text{Kr}[\text{twórz-sł.}'\text{Kowal}'].()) =$
 $(\text{'Kowal'}, (\text{'słowo'}))$

Stąd, zgodnie z definicją denotacji przypisania, drugie z trzech przypisań rozszerza typ rekordu o nowy atrybut `nazwisko` oraz nadaje mu wartość 'Kowal'. Analogicznie działa trzecie przypisanie.

Alternatywą do takiego postępowania jest zadeklarowanie zmiennej rekordowej o „docelowym” typie, a następnie nadanie jej wartości:

```
set typ-pracownik as
  record-type
    imię          of type word,
    nazwisko      of type word,
    rok-urodzenia of type number
  ee
tes ;
let pracownik as typ-pracownik tel ;
pracownik := record
                      imię          <= 'Jan',
```

```

nazwisko      <= 'Kowal',
rok-urodzenia <= 1968
ee

```

6.1.5 Instrukcja wymiany transferu

Ta instrukcja jest analogiczna do poprzedniej, tylko że tym razem wymianie podlega nie kompozyt wartości, ale jej transfer.

wymień-tr : Identyfikator x DenWyrTra $\mapsto$ DenIns

wymień-tr.(ide, tra-n).sta = (n – nowy)

jest-błąd.sta $\rightarrow$ sta

niech

((śrt, śrp), (wrt, 'OK')) = sta

wrt.ide = ? $\rightarrow$ 'niezadeklarowany-identyfikator'

niech

((kom, tra-d) = wrt.ide (d – dawny)

tra-n.kom $\neq$ (pp, ('boolowska')) $\rightarrow$ 'niespełnione-jarzmo'

niech

war-n = (kom, tra-n)

prawda $\rightarrow$ ((śrt, śrp), wrt[ide/war-n], 'OK')

Nowa wartość zachowuje poprzedni kompozyt, ale otrzymuje nowy transfer. Ten transfer musi być spełniony przez zastany kompozyt. Instrukcja wymiany transferu znajdzie swoje zastosowanie na gruncie **Lingua-SQL** (rozdział 12.7.6)⁷¹.

Konstruktor wymiany transfery umieściłem w grupie konstruktorów instrukcji, a nie deklaracji zmiennych dlatego, że może on być użyty w każdym miejscu programu, a nie jedynie w jego preambule (rozdział 6.1.9), jak deklaracje.

Zwracam uwagę, że w algebrze typów została zdefiniowana podobna instrukcja **wymień-ty-tr**, która jednak jest konstruktorem typów, a nie denotacją instrukcji.

6.1.6 Instrukcja trywialna

Instrukcja trywialna to identycznościowa transformacja stanu na stan. Jak zobaczymy dalej, posłuży ona do ujednolicenia deklaracji procedur funkcyjnych (rozdział 7.5). Denotację instrukcji trywialnej tworzymy niżej określonym konstruktorem:

twórz-ins-trywialną: $\mapsto$ DenIns

twórz-ins-trywialną().sta = sta

⁷¹ Tę bardzo ogólną postać modyfikacji transferu przyjąłem by nie komplikować mojego wykładu szczegółami technicznymi. W przypadku budowania praktycznego języka być może powinno się pomyśleć o szczególnych przypadkach modyfikacji, np. polegających na koniunktywnym wzbogaceniu transferu o nowy składnik.

6.1.7 Instrukcje strukturalne

Instrukcje strukturalne powstają przez użycie czterech konstruktorów. Trzy podstawowe to złożenie sekwencyjne, złożenie warunkowe i pętla. Czwarty służy do obsługi błędów. Rozpocznijmy od złożenia sekwencyjnego.

sekwencja-ins : DenIns x DenIns $\mapsto$ DenIns

sekwencja-ins.(din-1,din-2) = din-1 • din-2

Instrukcje złożone sekwencyjnie są wykonywane jedne po drugiej. Złożenie warunkowe definiujemy w sposób następujący:

jeżeli : DenWyrDan x DenIns x DenIns $\mapsto$ DenIns

jeżeli.(dwd, din-1, din-2).sta =

jest-błąd.sta $\rightarrow$ sta

dwd.sta = ? $\rightarrow$?

dwd.sta : Błąd $\rightarrow$ sta $\leftarrow$ dwd.sta

niech

(dan, kor) = dwd.sta

rodzaj.kor $\neq$ ('boolowska') $\rightarrow$ sta $\leftarrow$ 'oczekiwana-boolowska'

dan = pp $\rightarrow$ din-1.sta

prawda $\rightarrow$ din-2.sta

Zauważmy, że wobec obecności w naszym języku konstruktora pętli **dopóki** (patrz niżej) wykonanie każdej z instrukcji składowych może się nie zakończyć, co oznacza, że stany din-1.sta lub/i din-2.sta mogą być nieokreślone. Jeżeli więc dan = pp oraz din-1.sta jest nieokreślone, to nieokreślony jest stan końcowy instrukcji warunkowej, natomiast w przeciwnym przypadku, stan końcowy jest nieokreślony, gdy din-2.sta jest nieokreślone.

dopóki : DenWyrDan x DenIns $\mapsto$ DenIns

dopóki.(dwd, din).sta =

(1) jest-błąd.sta $\rightarrow$ sta

(2) dwd.sta = ? $\rightarrow$?

(3) dwd.sta : Błąd $\rightarrow$ sta $\leftarrow$ dwd.sta

niech

(dan, kor) = dwd.sta

(4) rodzaj.kor $\neq$ ('boolowska') $\rightarrow$ sta $\leftarrow$ 'oczekiwana-boolowska'

(5) dan = ff $\rightarrow$ sta

(6) **prawda** $\rightarrow$ (din • [dopóki.(dwd, din)]).sta

W definicji operatora **dopóki** mamy do czynienia z rekurencją, co oznacza, że ten operator jest zdefiniowany równaniem stałopunktowym. To za sprawą tego właśnie operatora wśród denotacji instrukcji mogą się pojawiać funkcje częściowe. W przyszłości, gdy wprowadzimy procedury funkcyjne i imperatywne (rozdział 7), częściowość denotacji **dopóki**.(dwd, din) będzie mogła mieć trzy różne źródła:

1. gdy w wyrażeniu sterującym pętli **dwd** znajduje się wywołanie procedury funkcyjnej powodującej nieokreśloność, a więc gdy **dwd.sta = ?**,
2. gdy w instrukcji znajduje się lokalne **dopóki**, które powoduje nieokreśloność lub wywołanie procedury funkcyjnej lub imperatywnej z podobnym skutkiem, a więc gdy **din.sta = ?**
3. gdy nastąpi zapętlenie na poziomie całej instrukcji.

Komentarz 6.1.7-1 Definicja konstruktora **dopóki** ma charakter rekurencyjny, gdyż ten operator występuje w ciele swojej definicji. Mamy więc do czynienia z równaniem stałopunktowym w zbiorze łańcuchowo-zupełnym funkcji częściowych (por. rozdział 2.6), którymi są denotacje instrukcji:

DenIns = Stan → Stan

Jednakże to nie konstruktor **dopóki** jest rozwiązaniem równania stałopunktowego, ale wynik jego zastosowania do pary (**dwd, din**), a więc funkcja

dopóki.(dwd, din) : Stan → Stan

I oczywiście dla każdej pary (**dwd, din**) mamy do czynienia z innym takim równaniem. Aby mieć pewność, że rozwiązanie każdego z tych równań istnieje, trzeba pokazać, że prawa strona każdego z nich reprezentuje operator ciągły w zbiorze łańcuchowo-zupełnym **Stan → Stan**. W tym celu wprowadzę następujące oznaczenia:

NieOK = {(sta, sta) | spełnione (1)}

BIWyr = {(sta, sta ◀ **dwd.sta**) | spełnione (3) oraz nie (1) i (2)}

NieBoo = {(sta, sta ◀ 'oczekiwana-boolowska') | spełnione (4) oraz nie (1), (2) i (3)}

FF = {(sta, sta) | spełnione (5) oraz nie (1), (2), (3) i (4)}

TT = {(sta, sta) | nie (1), (2), (3), (4) i (5)}

Naszą definicję możemy teraz zapisać jako równanie stałopunktowe:

X = NieOK | BIWyr | NieBoo | FF | TT • din • X

Ponieważ występujące w tym równaniu operatory „|” oraz „•” są ciągłe, to jego najmniejsze rozwiązanie istnieje, a ponieważ współczynniki równania mają wzajemnie rozłączne dziedziny, to z Tw.2.6-1 wynika, że to rozwiązanie jest funkcją postaci:

X = (TT • Din)* • (NieOK | BIWyr | NieBoo | FF)

6.1.8 Obsługa błędów

Ostatni konstruktor strukturalny związany jest z *obsługą błędów*. Pozwala on na uruchomienie wskazanej instrukcji w przypadku pojawienia się wskazanego błędu.

jeżeli-błąd : DenWyrDan x DenIns → DenIns

jeżeli-błąd.(dwd, din).sta =

nie jest-błąd.sta → sta

niech

(śro, (wrt, bld)) = sta

sta-1 = sta ◀ 'OK'

dwd.sta-1 = ? → ?

niech

kom = dwd.sta-1

kom : Błąd → sta ◀ kom @ 'obsługa-błędu-nie-została-wykonana'

rodzaj.kom $\neq$ ('słowo') $\rightarrow$ 'sta $\leftarrow$ 'oczekiwane-słowo' @
'obsługa-błędu-nie-została-wykonana'

niech

(sło, ('słowo')) = kom

sło $\neq$ bld $\rightarrow$ sta

din.sta-1 = ? $\rightarrow$?

niech

sta-2 = din.sta-1

jest-błąd.sta-2 $\rightarrow$ sta $\leftarrow$ błąd.sta-2 @ 'obsługa-błędu-nie-została-wykonana'

prawda $\rightarrow$ sta-2 $\leftarrow$ bld @ 'obsługa-błędu-została-wykonana'

Jeżeli stan początkowy nie niesie błędu, to staje się on stanem wyjściowym.

W przeciwnym przypadku zostaje utworzony stan tymczasowy Sta-1, który powstaje przez usunięcie komunikatu błędu ze stanu początkowego. W nowym stanie jest wyliczana wartość wyrażenia o denotacji dwd, które ma wskazać obsługiwany błąd. Jeżeli to wyliczenie się nie zakończy, to nie zakończy się również wykonanie całej instrukcji. Jeżeli w wyniku tego wyliczenia pojawi się błąd lub kompozyt, który nie niesie słowa, to jest generowany odpowiedni komunikat błędu uzupełniony o informację, że obsługa błędu nie została wykonana.

W przeciwnym przypadku, jeżeli słowo niesione przez kompozyt — czyli błąd, który ma być obsłużony — różni się od komunikatu błędu w stanie początkowym, to stan końcowy jest ponownie identyczny z początkowym.

W przeciwnym przypadku w stanie tymczasowym jest wykonywana instrukcja będąca argumentem konstruktora. Jeżeli w wyniku tego wykonania pojawi się błąd, to jest on sygnalizowany wraz z komentarzem, że obsługa błędu nie została wykonana.

W przeciwnym przypadku do stanu wyjściowego dla din zostaje załadowany początkowy komunikat błędu uzupełniony o informację, że obsługa błędu została wykonana.

Jak więc widzimy, wyrażenie, które występuje w instrukcji obsługi błędu musi jako swoją wartość przyjmować słowo. Jeżeli to słowo jest identyczne z aktualnie wygenerowanym komunikatem błędu, to jest wykonywana „wewnętrzna” instrukcja instrukcji obsługi błędu.

Podaną tu definicję należy traktować jako przykład pokazujący jedynie, że obsługa błędów da się w naszym modelu opisać. W żadnym wypadku nie należy jej traktować jako wzorzec dla takiej obsługi. Inne przykłady mechanizmów obsługi błędów zostaną pokazane w rozdziałach 7.3.3 oraz 12.7.6.4.

6.1.9 Preambuły i programy

Preambuły są sekwencjami „dowolnie wymieszanych” definicji stałych typologicznych, deklaracji zmiennych danologicznych i preambuły trywialnej. Ich denotacje tworzymy za pomocą czterech konstruktorów:

twórz-preambułę-z-def-typu	: DenDefTyp	$\mapsto$ DenPre
twórz-preambułę-z-dek-zmiennej	: DenDekZmi	$\mapsto$ DenPre
twórz-pre-trywialną	:	$\mapsto$ DenPre

sekwencja-pre : DenPre x DenPre $\mapsto$ DenPre

Pierwsze dwa konstruktory są identycznościowymi włożeniami, tj. funkcjami identycznościowymi, które pozwalają nam potraktować element jednego nośnika algebry jako element innego. Na gruncie algebry denotacji ta konstrukcja oznacza, że każda definicja typu i każda deklaracja zmiennej może być „potraktowana” jako preambuła. Zauważmy, że bez tych konstruktorów część osiągalna nośnika denotacji preambuł byłaby pusta, a wraz z nią pusta byłaby część osiągalna nośnika programów. To z kolei oznaczałoby, że puste byłyby odpowiadające im nośniki algebry składni.

Uwaga! Puste byłyby (składniowe) nośniki preambuł i programów, a nie jedynie ich części osiągalne, bo algebra składni jest zawsze osiągalna. Innymi słowy za pomocą gramatyki równaniowej naszej składni nie dawałoby się wygenerować ani preambuł ani programów.

Trzeci konstruktor tworzy funkcję identycznościową na stanach analogicznie jak w przypadku instrukcji (rozdział 6.1.6):

twórz-pre-trywialną().sta = sta

Ten konstruktor ma również charakter jedynie techniczny i pozwala na zdefiniowanie programu jako pary składającej się z preambuły, być może trywialnej, po której następuje instrukcja, również być może trywialna.

twórz-program : DenPre x DenIns $\mapsto$ DenPro

twórz-program.(dep, ins) = dep • ins

Program z preambułą trywialną, jeżeli zostanie wykonany „bez kontekstu”, to oczywiście zawsze wygeneruje błąd. Dopuszczam jednak takie programy, gdyż w **Lingua-2** będą one mogły występować jako treści procedur. Z kolei programy z trywialną zarówno preambułą jak i instrukcją będą mogły występować w deklaracjach procedur funkcyjnych⁷².

6.1.10 Podsumowanie dotyczące roli typów w programach

Każda występująca w programie stała typologiczna jest zadeklarowana tylko raz i pozostaje niezmienną na czas wykonywania programu. Dlatego identyfikatory związane w środowisku typów określam jako identyfikatory stałych typologicznych. Te stałe są wykorzystywane zarówno w deklaracjach zmiennych jak i w definicjach kolejnych typów. W tym drugim przypadku służą do budowania złożonych typów metodą wstępującą (bottom-up).

Każda występująca w programie zmienna danologiczna jest w programie zadeklarowana tylko raz, a dokonuje się tego przez przypisanie jej pseudowartości (Ω , (kor, tra)). W trakcie wykonywania programu typ tej zmiennej początkowo ustalony jako (kor, tra) może ulegać zmianie jedynie w dwóch przypadkach:

- przez instrukcję przypisania, która zmienia korpus na nowy, ale koherentny z poprzednim; na razie ta sytuacja może mieć miejsce jedynie w przypadku zmiennych rekordowych, natomiast w **Lingua-SQL** zostanie rozszerzona na zmienne wierszowe i tabelowe,
- przez instrukcję zmiany transferu, ale wtedy nowy transfer musi być spełniony przez zastany kompozyt.

⁷² Oba rozwiązania, choć w nieco innej wersji, zasugerował mi Andrzej Tarlecki.

6.2 Składnia

Założenie, że **Lingua-1** stanowi rozszerzenie **Lingua-A** dotyczy oczywiście nie tylko algebry denotacji, ale składni abstrakcyjnej, konkretnej i kolokwialnej. Zatem w dalszych rozdziałach opiszę jedynie te elementy składni **Lingua-1**, które nie występują w **Lingua-A**.

6.2.1 Składnia abstrakcyjna

Deklaracje zmiennych

```
dez : DekZmiA =
    deklaruj-zmi-dan (Identyfikator , WyrTypA) |
    sekwencja-dez (DekZmiA ; DekZmiA)
```

Definicje typów

```
det : DefTypA =
    definiuj-sta-typ (Identyfikator , WyrTypA) |
    sekwencja-det (DefTypA ; DefTypA)
```

Instrukcje

```
ins : InstrukcjaA =
    przypisz (Identyfikator , WyrDanA) |
    wymień-tr (Identyfikator, WyrTra) |
    twórz-ins-trywialną () |
    jeżeli (WyrDanA , InstrukcjaA , InstrukcjaA) |
    jeżeli-błąd (WyrDanA , InstrukcjaA) |
    dopóki (WyrDanA , InstrukcjaA) |
    sekwencja-ins (InstrukcjaA , InstrukcjaA)
```

Preambuły

```
pab : PreambułaA =
    twórz-preambułę-z-def-typu (DefTypA) |
    twórz-preambułę-z-dek-zmiennej (DekZmiA) |
    twórz-pre-trywialną () |
    sekwencja-pre (PreambułaA , PreambułaA)
```

Programy

```
prg : ProgramA =
    twórz-program (PreA , InsA)
```

6.2.2 Składnia konkretna

Składnię konkretną części aplikatywnej przyjmuję za **Lingua-A**. Nowe konkretyzacje opisuje niżej przedstawiona gramatyka:

Deklaracje zmiennych

```
dez : DekZmi =
  let Identyfikator be WyrTyp tel |
    DekZmi ; DekZmi
```

Definicje typów

```
det : DefTyp =
  set Identyfikator as WyrTyp tes |
    DefTyp ; DefTyp
```

Instrukcje

```
ins : Instrukcja =
  Identyfikator := WyrDan |
  yoke Identyfikator := WyrTra |
  skip |
  if WyrDan then Instrukcja else Instrukcja fi |
  if-error WyrDan then Instrukcja fi |
  while WyrDan do Instrukcja od |
  Instrukcja ; Instrukcja
```

Preambuły

```
pab : Preambuła =
  DefTyp |
  DekZmi |
  skip |
  Preambuła ; Preambuła
```

Programy

```
prg : Program =
  begin-program Preambuła ; Instrukcja end-program
```

Jedyną zmianą pomiędzy składniami abstrakcyjną i konkretną, która jest istotnie homomorficzna, tj. sklejająca, a więc nieizomorficzna, jest opuszczenie nawiasów przy operacji składania sekwencyjnego deklaracji zmiennych, definicji typów, instrukcji i preambuł.

W tym miejscu czytelnikowi należą się jednak wyjaśnienia dotyczące „pozornego sklejania” w dwóch przypadkach:

1. użycia tego samego kontekstu **if then else fi** zarówno w przypadku wyrażeń warunkowych, jak i warunkowych instrukcji,
2. użycia tego samego słowa kluczowego **skip** zarówno w przypadku instrukcji, jak i preambuł.

Mimo, że w obu tych przypadkach mamy do czynienia z pewnym rodzajem sklejania, to nie uniemożliwia ono zbudowania semantyki denotacyjnej (por. twierdzenie 2.13-1 w rozdziale 2.13), bowiem taka sytuacja może mieć miejsce jedynie wtedy, gdy homomorfizm odwzorowuje dwie frazy abstrakcyjne tego samego nośnika w jedną konkretną. W obu opisanych wyżej przypadkach tak jednak nie jest.

Fakt, że sklejanie „międzyrodzajowe” nie wyklucza istnienia semantyki denotacyjnej praktycznie oznacza, że w takim przypadku parser — który ma odwzorować składnię konkretną w abstrakcyjną — rozpozna rodzaj wieloznacznego napisu, np. czy **skip** jest instrukcją czy preambułą, na podstawie kontekstu⁷³.

Oczywiście na poziomie składni abstrakcyjnej musieliśmy wprowadzić po dwa różne napisy, bo odpowiadały one różnym funkcjom algebry denotacji.

Nasza gramatyka konkretna jest jednak wieloznaczna, a to dzięki opuszczeniu nawiasów związanych z operatorem składania sekwencyjnego „;”. Należy więc pokazać, że algebra składni konkretnej nie jest bardziej wieloznaczna od algebry denotacji. Dowód opiera się oczywiście na fakcie, że średniki odpowiadają sekwencyjnemu łączeniu funkcji, które jest operacją łączną. Jednakże proste stwierdzenie tego faktu za dowód nie wystarcza, bo przecież dodawanie i mnożenie również są łączne, ale nie „względem siebie” więc w formule $((x + y) + z) * w$ można by opuścić tylko niektóre nawiasy.

Formalnie rzecz ujmując, trzeba udowodnić, że nasza semantyka abstrakcyjna i konkretna spełniają implikację (2.13-1) z rozdziału 2.13. Taki dowód musi przebiegać przez indukcję strukturalną względem gramatyki składni abstrakcyjnej. Poniżej ograniczę się do jego szkicu lub — jak mawiają niektórzy — agitacji za dowodem, który przeprowadzę na przykładzie instrukcji. Rozpocznę od wprowadzenia trzech oznaczeń:

A_{in}	: InstrukcjaA	$\mapsto$ Instrukcja	— nasz homomorfizm ze składni abstrakcyjnej w konkretną dla instrukcji
A_{wd}	: WyrDanA	$\mapsto$ WyrDan	— nasz homomorfizm ze składni abstrakcyjnej w konkretną dla wyrażeń,
D_{in}	: InstrukcjaA	$\mapsto$ DenIns	— jednoznacznie wyznaczony homomorfizm będący semantyką instrukcji abstrakcyjnych.

Należy pokazać, że dla każdych dwóch abstrakcyjnych instrukcji

$ins-1, ins-2 : InstrukcjaA$

zachodzi następująca implikacja:

jeżeli

$$A_{in}.[ins-1] = A_{in}.[ins-2] \tag{1}$$

to

⁷³ Formalnie rzecz biorąc w miejsce **skip** można by użyć spacji, jednakże w mojej opinii **skip** jest wyborem bezpieczniejszym z tego powodu, że programista musi to słowo świadomie wpisać, podczas gdy spację może pozostawić w programie przez przeoczenie. W pierwszym przypadku parser analizujący instrukcję postaci **if** $x > 0$ **then** $x := x + 1$ **else** **fi** będzie sygnalizował błąd, w drugim potraktuje ją jak **if** $x > 0$ **then** $x := x + 1$ **else** **skip** **fi**, co mogłoby być niezgodne z intencją programisty.

$$\text{Din.}[\text{ins-1}] = \text{Din.}[\text{ins-2}] \quad (2)$$

W kroku pierwszym indukcji zauważmy, że jeżeli ins-1 jest przypisaniem, to z równości (1) wynika, że $\text{ins-1} = \text{ins-2}$, a więc (2) zachodzi. Załóżmy więc, że

$$\text{ins-1} = \text{jeżeli}(\text{wda-1}, \text{ins-11}, \text{ins-12})$$

Wtedy

$$\text{Ain.}[\text{ins-1}] = \text{if Awd.}[\text{wda-1}] \text{ then Ain.}[\text{ins-11}] \text{ else Ain.}[\text{ins-12}] \text{ fi}$$

a więc na mocy (1) muszą istnieć takie wda-2 , ins-21 , ins-22 , że jest spełniona równość

$$\text{Ain.}[\text{ins-2}] = \text{if Awd.}[\text{wda-2}] \text{ then Ain.}[\text{ins-21}] \text{ else Ain.}[\text{ins-22}] \text{ fi}$$

oraz że są spełnione równości

$$\text{Awd.}[\text{wda-1}] = \text{Awd.}[\text{wda-2}]$$

$$\text{Ain.}[\text{ins-11}] = \text{Ain.}[\text{ins-21}]$$

$$\text{Ain.}[\text{ins-12}] = \text{Ain.}[\text{ins-22}]$$

a więc z założenia indukcyjnego musi zachodzić równość $\text{Din.}[\text{ins-1}] = \text{Din.}[\text{ins-2}]$. Analogiczne dowody można przeprowadzić dla pozostałych konstruktorów poza sekwencja-ins . W tym bowiem przypadku z równości:

$$\text{Ain.}[\text{sekwencja-ins}(\text{ins-11}, \text{ins-12})] = \text{Ain.}[\text{sekwencja-ins}(\text{ins-21}, \text{ins-22})]$$

nie można w przypadku ogólnym wnioskować, że

$$\text{Ain.}[\text{ins-11}] = \text{Ain.}[\text{ins-21}] \text{ oraz}$$

$$\text{Ain.}[\text{ins-12}] = \text{Ain.}[\text{ins-22}],$$

a więc nie można skorzystać z założenia indukcyjnego. Rozpatrzmy więc podprzypadki tego przypadku. Niech pierwszy dotyczy sytuacji, gdy ins-11 nie jest postaci

$$\text{sekwencja-ins}(\dots)$$

Wtedy z (1) wynika, że tej postaci nie może być też ins-21 , co pozwala nam skorzystać z założenia indukcyjnego. Niech więc:

$$\text{ins-1} = \text{sekwencja}(\text{sekwencja-ins}(\text{ins-111}, \text{ins-112}), \text{ins-12})$$

Stąd i z (1)

$$\text{Ain.}[\text{ins-1}] = \text{Ain.}[\text{ins-111}] ; \text{Ain.}[\text{ins-112}] ; \text{Ain.}[\text{ins-12}]$$

$$\text{Ain.}[\text{ins-2}] = \text{Ain.}[\text{ins-211}] ; \text{Ain.}[\text{ins-212}] ; \text{Ain.}[\text{ins-22}]$$

oraz

$$\text{Ain.}[\text{ins-111}] = \text{Ain.}[\text{ins-211}]$$

$$\text{Ain.}[\text{ins-112}] = \text{Ain.}[\text{ins-212}]$$

$$\text{Ain.}[\text{ins-12}] = \text{Ain.}[\text{ins-22}]$$

a więc z założenia indukcyjnego

$$\text{Din.}[\text{ins-111}] = \text{Din.}[\text{ins-211}]$$

$$\text{Din.}[\text{ins-112}] = \text{Din.}[\text{ins-212}]$$

$$\text{Din.}[\text{ins-12}] = \text{Din.}[\text{ins-22}]$$

Teraz mamy do rozważenia dwa przypadki. Pierwszy to sytuacja gdy

$$\text{ins-2} = \text{sekwencja-ins}(\text{sekwencja-ins}(\text{ins-211}, \text{ins-212}), \text{ins-22})$$

Wtedy na mocy założenia indukcyjnego możemy wnioskować o spełnieniu (2). Drugi przypadek to sytuacja, w której:

$$\text{ins-2} = \text{sekwencja-ins}(\text{ins-211}, \text{sekwencja-ins}(\text{ins-212}, \text{ins-22}))$$

Wtedy

$$\text{Din.}[\text{ins-1}] = (\text{Din.}[\text{ins-111}] \bullet \text{Din.}[\text{ins-112}]) \bullet \text{Din.}[\text{ins-12}]$$

$$\text{Din.}[\text{ins-2}] = \text{Din.}[\text{ins-211}] \bullet (\text{Din.}[\text{ins-212}] \bullet \text{Din.}[\text{ins-22}])$$

Teraz oczekiwana teza dla instrukcji wynika z założenia indukcyjnego oraz z faktu, że operacja sekwencyjnego łączenia funkcji jest łączna. Dla pozostałych kategorii języka dowód przebiega analogicznie.

Z punktu widzenia użytkownika języka zasada łączenie preambuł, deklaracji, definicji i instrukcji średnikami bez używania nawiasów jest bardzo prosta, jednakże z punktu widzenia budowniczego implementacji prowadzi do konieczności zbudowania algorytmu, który będzie w jednoznaczny sposób zamieniał programy napisane w składni konkretnej, a więc bez nawiasów odpowiadających średnikom, na programy w składni abstrakcyjnej, a więc z nawiasami. W tym przypadku można przyjąć dowolną strategię, np. nawiasowania od lewej do prawej, co oznacza, że instrukcja:

`ins-1 ; ins-2 ; ins-3`

zostanie przekształcona na

`(ins-1 ; (ins-2 ; ins-3))`

W tym miejscu warto dodać komentarz o charakterze historycznym. Otóż w prehistorycznych czasach, gdy podręczniki języków programowania zawierały gramatyki opisujące ich składnię, budowano jedną tylko gramatykę — wspólną dla użytkownika i twórcy implementacji. W konsekwencji użytkownikowi przedstawiano tę, z której dawało się łatwo wygenerować parser. Niestety jej złożoność zniechęcała użytkowników do posługiwania się nimi, co spowodowało, że z biegiem lat w ogóle zrezygnowano z formalnych definicji składni języka zastępując je przykładami. Wylano więc dziecko z kąpielą i doprowadzono do sytuacji, o których pisałem na początku książki.

Komentarz 6.2.2-1 Tworząc składnię konkretną dla **Lingua-1** zastosowałem rozwiązania znane mi z dawnych czasów, które — mimo iż nie są dziś powszechne — uważam za praktyczne z punktu widzenia czytelności programów, co najczęściej przekłada się też na mniejszą liczbę błędów popełnianych przez programistów. Oto one:

1. Znak przypisania „:=” nie jest równością, jak w niektórych językach, ale odrębnym znakiem.
2. Zastosowałem nawiasy zamykające **fi** dla **if** i **while** gdyż moje doświadczenie wskazuje, że to ułatwia budowanie graficznie jasnej struktury programu.
3. Hierarchicznie tworzone wcięcia i spacje ułatwiają wyeksponowanie struktury programu, gdy jednak grają rolę nawiasów (np. jak w języku Python), to łatwo o błąd wynikający np. z pomyłkowego naciśnięcia klawisza **Del**. Jako matematyk buntuję się też przeciw zasadzie, by niewidoczny znak formatujący był elementem składni. Wygodnie jest jednak używać przełamań wiersza i wcięć w sposób dowolny, tj. nie zaburzający znaczenia programu. W **Lingua** są one usuwane przez transformację przywracającą.

6.2.3 Składnia kolokwialna

Do kolokwializmów znanych już nam z **Lingua-A** dodaję jedynie cztery nowe zasady. Po pierwsze, w definicjach typów i deklaracje zmiennych, gdzie występują jarzma trywialne **true**, możemy zastosować notację skróconą. Na przykład zamiast definicji typu

```
set list-of-names as type list-of word ee with true tes
```

napiszemy

```
set list-of-names as list-of word ee tes
```

i analogicznie zamiast deklaracji zmiennej

```
let names be list-of-names ee with true tel
```

napiszemy

```
let names be list-of-names tel
```

Po drugie, deklaracje zmiennych tego samego typu mogą być grupowane w jedną deklarację wielu zmiennych, np. w miejsce

```
let x be number tel;
```

```
let y be number tel;
```

```
let z be number tel
```

napiszemy

```
let x, y, z be number tel
```

i analogicznie dla definicji stałych typologicznych.

Po trzecie, w tekst programu można wpisywać komentarze, które są usuwane przez transformację przywracającą. Każdy komentarz zaczyna się od znaku „#” i kończy przełamaniem wiersza.

Po czwarte, w programie możemy dowolnie używać przełamań wiersza i spacji. Wszystkie zostaną usunięte przez transformację przywracającą.

6.2.4 Przykład prostego programu

Oto przykład programu tworzącego prostą bazę danych osobowych w postaci tablicy rekordów pracowniczych. Pod każdym elementem programu umieszczam jego opis.

```
set typ_wykaz as  
  array-type word ee  
tes
```

Identyfikator **typ_wykaz** zostaje zadeklarowany jako stała typologiczna typu tablicowego ((**T**, (**'słowo'**)), **PP**), gdzie **PP** jest denotacją dla wyrażenia transferowego **true**.

```
set typ_pracownik as  
  record-type  
    imię, nazwisko of type word  
    rok_urodzenia of type real ,  
    odznaczenia of type typ_wykaz
```

```
ee
```

```
tes;
```

Identyfikator `typ_pracownik` zostaje zadeklarowany jako stała typologiczna typu rekordowego ('R', ['imię' / ('słowo'), ...], PP)

```
set typ_baza_osobowa as
```

```
  array-type typ_pracownik ee
```

```
tes
```

Identyfikator `typ_baza_osobowa` zostaje zadeklarowany jako stała typologiczna typu tablicowego ('T', ('R', ['imię' / ('słowo'), ...], PP)

```
let baza_sprzedawców be typ_baza_osobowa ee
```

```
let sprzedawca be typ_pracownik ee
```

```
let odznaczenia_Kowal be typ_wykaz ee
```

```
let odznaczenie_1, odznaczenie_2, odznaczenie_3 be word ee
```

Cztery identyfikatory zostały zadeklarowane jako zmienne danologiczne, którym w wartościowaniu przypisano pseudodane ze wskazanymi typami.

```
odznaczenie_1 := 'wyróżniający się pracownik zaplecza'
```

```
odznaczenie_2 := 'znakomity sprzedawca'
```

```
odznaczenie_3 := 'gwiazda sprzedaży'
```

Trzem zmiennym danologicznym zostały przypisane w wartości typu ('słowo').

```
odznaczenia_Kowal :=
```

```
  array [odznaczenie_1, odznaczenie_2, odznaczenie_3]
```

Zmiennej danologicznej została przypisany w wartościowaniu wartość tablicowa typu (('T', ('słowo')), PP). Zauważmy, że zmiennej `odznaczenia_Kowal` został przypisany w typowaniu ten typ, który został przypisany w środowisku stałej typologicznej `typ_wykaz`, a więc typ (('T', ('słowo')), PP).

```
sprzedawca :=
```

```
  record
```

```
    imię          <= 'Jan'
```

```
    nazwisko      <= 'Kowal'
```

```
    rok-urodzenia <= 1968
```

```
    odznaczenia   <= odznaczenia_Kowal
```

```
  ee
```

Zmiennej danologicznej `sprzedawca` został przypisana wartość rekordowa typu przypisanego stałej typologicznej `typ_pracownik`.

```
baza_sprzedawców := array pracownik [sprzedawca]
```

Zmiennej danologicznej `baza_sprzedawców` została przypisana jednoelementowa tablica typu przypisanego stałej typologicznej `baza_osobowa`. Jedyne element tej tablicy jest utworzonym wcześniej rekordem typu `typ_pracownik`.

6.3 Semantyka

Na definicję semantyki języka **Lingua-1** składa się omówiona w rozdziale 5.7 definicja semantyki dla **Lingua-A** powiększona o klauzule definicyjne konstrukcji imperatywnej części języka:

$$\begin{aligned} \text{Sdz} : \text{DekZmi} &\mapsto \text{DenDekZmi} \\ \text{Sdt} : \text{DefTyp} &\mapsto \text{DenDefTyp} \\ \text{Sin} : \text{Instrukcja} &\mapsto \text{DenIns} \\ \text{Spre} : \text{Preambuła} &\mapsto \text{DenPre} \\ \text{Spr} : \text{Program} &\mapsto \text{DenPro} \end{aligned}$$

Definicje tych semantyk podam w wersji algebraicznej (rozdział 5.6), ale za to przytoczę je wszystkie.

Deklaracje zmiennych danologicznych

$$\begin{aligned} \text{Sdz} : \text{DekZmi} &\mapsto \text{DenDekZmi} && \text{czyli} \\ \text{Sdz} : \text{DekZmi} &\mapsto \text{Stan} \mapsto \text{Stan} \\ \\ \text{Sdz}[\text{let } \text{id} \text{ be } \text{wyt}] &= \text{zmienna-dan}(\text{Sid}[\text{id}], \text{Swty}[\text{wyt}]) \\ \text{Sdt}[\text{dez-1}; \text{dez-2}] &= \text{sekwencja-def}(\text{Sdt}[\text{dez-1}], \text{Sdt}[\text{dez-2}]) \end{aligned}$$

Definicje stałych typologicznych

$$\begin{aligned} \text{Sdt} : \text{DefTyp} &\mapsto \text{DenDefTyp} \\ \text{Sdt} : \text{DefTyp} &\mapsto \text{Stan} \mapsto \text{Stan} \\ \\ \text{Sdt}[\text{set } \text{id} \text{ as } \text{wyt}] &= \text{definiuj-sta-typ}(\text{Sid}[\text{id}], \text{Swty}[\text{wyt}]) \\ \text{Sdt}[\text{det-1}; \text{det-2}] &= \text{sekwencja-def}(\text{Sdt}[\text{det-1}], \text{Sdt}[\text{det-2}]) \end{aligned}$$

Instrukcje

$$\begin{aligned} \text{Sin} : \text{Instrukcja} &\mapsto \text{DenIns} \\ \text{Sin} : \text{Instrukcja} &\mapsto \text{Stan} \rightarrow \text{Stan} \\ \\ \text{Sin}[\text{id} := \text{wyr}] &= \text{przypisz}(\text{Sid}[\text{id}], \text{Sw}[\text{wyr}]) \\ \text{Sin}[\text{if } \text{wyd} \text{ then } \text{ins-1} \text{ else } \text{ins-2} \text{ fi}] &= \text{jeżeli}(\text{Sw}[\text{wyd}], \text{Si}[\text{ins-1}], \text{Si}[\text{ins-2}]) \\ \text{Sin}[\text{while } \text{wyd} \text{ do } \text{ins} \text{ od}] &= \text{dopóki}(\text{Sw}[\text{wyd}], \text{Si}[\text{ins}]) \\ \text{Sin}[\text{ins-1}; \text{ins-2}] &= \text{sekwencja-ins}(\text{Si}[\text{ins-1}], \text{Si}[\text{ins-2}]) \end{aligned}$$

Preambuły

$\text{Spre} : \text{Preambuła} \mapsto \text{DenPre}$

$\text{Spre} : \text{Preambuła} \mapsto \text{Stan} \rightarrow \text{Stan}$

$\text{Spre}[\text{det}] = \text{Sdt}[\text{det}]$

$\text{Spre}[\text{dez}] = \text{Sdz}[\text{dez}]$

$\text{Spre}[\text{pab-1}; \text{pab-2}] = \text{sekwencja-pre}(\text{Spre}[\text{pab-1}], \text{Spre}[\text{pab-2}])$

Programy

$\text{Spr} : \text{Program} \mapsto \text{DenPro}$

$\text{Spr} : \text{Program} \mapsto \text{Stan} \rightarrow \text{Stan}$

$\text{Spr}[\text{ins}] = \text{twórz-program-z-instrukcji}(\text{Sin}[\text{ins}])$

$\text{Spr}[\text{pab}; \text{ins}] = \text{twórz-program}(\text{Spre}[\text{pab}], \text{Sin}[\text{ins}])$

7 Lingua-2 — procedury

7.1 Wprowadzenie do modelu dla procedur

7.1.1 O procedurach historycznie

Pojęcie procedury pojawiło się w językach programowania już w latach 1950. Początkowo były to proste listy instrukcji zwane *podprogramami*, które komunikowały się z programem głównym jedynie za pomocą zmiennych globalnych. Z czasem, dla zwiększenia uniwersalności procedur, zaopatrzono je w mechanizm przyjmowania parametrów, zwanym kolokwialnie *wołaniem parametrów* (ang. *parameter call*). Wprowadzono też możliwość deklarowania zmiennych lokalnych dostępnych (widocznych) jedynie wewnątrz procedury.

Obok tak rozumianych procedur, które dalej będę nazywał *procedurami imperatywnymi*, nieco później wprowadzono *procedury funkcyjne*, które semantycznie odpowiadały wyrażeniom, bowiem ich wykonanie generowało nie stan końcowy, ale wartość.

Najpowszechniej używanym proceduralnym językiem wyższego rzędu z przełomu lat 1950/1960. był Fortran. W tym języku procedury mogły wywoływać inne procedury, ale nie same siebie. Ta ostatnia konstrukcja została wprowadzona w języku Algol 60, gdzie określono ją mianem *procedur rekurencyjnych*. Twórcy Algolu 60 w wykorzystaniu procedur poszli jeszcze dalej, zezwalając nie tylko na rekurencję, ale także i na to, aby procedury mogły przyjmować inne procedury, a także same siebie, jako parametry. Pisałem o tym w rozdziale 4.1. Samoaplikacja procedur została jednak dość szybko porzucona i nie pojawiała się już w późniejszych językach. Natomiast procedury rekurencyjne, jako bardzo wygodny mechanizm programowania, są stosowane do dziś. W niektórych językach pozostawiono też konstrukcję pozwalającą na powoływanie procedur jako parametrów, ale nie samych siebie. Więcej na ten temat w rozdziale 7.6.

W tym miejscu warto wspomnieć, że na przełomie dekad 1950 i 1960 powstał w Polsce język porównywalny z ówczesnym Fortranem o nazwie SAKO (System Automatycznego Kodowania). Jego kompilator został uruchomiony na komputerze polskiej konstrukcji XYZ, który zbudowano w Zakładzie Aparatów Matematycznych przy Instytucie Matematyki PAN. Był to pierwszy komputer na jakim uczyłem się programowania jeszcze jako student. Jego początkowa wersja jako jedyną pamięć miała pamięć operacyjną o pojemności 1024 słów (dyski nie były jeszcze znane), czyli 1KB, wejście stanowił czytnik kart dziurkowanych, a wyjście — urządzenie dziurkujące karty. Aby wydrukować treści zapisane na kartach przez komputer, jechało się do Głównego Urzędu Statystycznego, gdyż tylko tam mieli potrzebne urządzenie. Później dodano pamięć magnetyczną w postaci bębna, której pojemność wynosiła chyba 5 KB.

Z czasów kiedy uczyłem się programowania w SAKO pamiętam dwa zalecenia dawane nam przez doświadczonych programistów. Pierwsze brzmiało, aby każda linia programu rozpoczynała się od etykiety, a kończyła instrukcją skoku **goto** wskazującą etykietę następnej instrukcji do wykonania. W tamtym czasie instrukcje skoku stanowiły jedyny mechanizm pozwalający na tworzenie rozgałęzień i pętli. Instrukcje typu **if-then-else** oraz **while** wprowadzono

później. Dlaczego jednak zalecano wstawiać **goto** na końcu każdej instrukcji? Przecież już wtedy obowiązywała zasada, że instrukcje wykonujemy w kolejności ich ustawienia w programie. Otóż program zadawano komputerowi w postaci pakietu kart dziurkowanych, z których każda zawierała jedną instrukcję. Kolejność kart wyznaczała więc kolejność wykonywania instrukcji. Jeżeli więc taki pakiet zawierający kilkaset instrukcji wymuszał się programiście z ręki, to poskładanie ich w prawidłowej kolejności wymagało ogromnej pracy. Jednakże, gdy kolejności wykonania była określona przez instrukcje skoku, karty można było wkładać do czytnika w dowolnym porządku.

Drugie zalecenie mówiło, że każdy program powinien mieścić się na jednej kartce formatu A4. Wyrażało ono przekonanie, że aby dobrze zrozumieć strukturę programu, powinien on dać się ogarnąć jednym spojrzeniem. W przypadku dużych programów takie zalecenie dawało się zrealizować jedynie dzięki mechanizmowi podprogramów, a później procedur. Pierwsza kartka zawiera wywołania głównych procedur programu. Definicje tych procedur, które mogły zawierać wywołania kolejnych procedur, zapisywano na kolejnych kartkach. Ten sposób tworzenia programów był charakterystyczny dla tzw. *programowania strukturalnego* (ang. *structured programming*) o którym pisałem już w rozdziale 4.1. Była to odpowiedź na problemy związane z licznymi błędami w programach pisanych wcześniej w postaci długich list zawierających wiele tysięcy instrukcji bez wewnętrznej struktury.

Programowanie strukturalne miało też doprowadzić inżynierię oprogramowania do stanu, w którym twórca dużego programu budowałby go z wcześniej utworzonych składowych o znanych i zagwarantowanych cechach funkcjonalnych. Taki sposób postępowania pozwalałby oprzeć dowód poprawności programu — a więc dowód jego zgodności z oczekiwaniami użytkownika — na wcześniej przeprowadzonych dowodach poprawności jego składowych. Niestety, jak na razie, ten cel nie został do końca osiągnięty (por. rozdział 3.1)

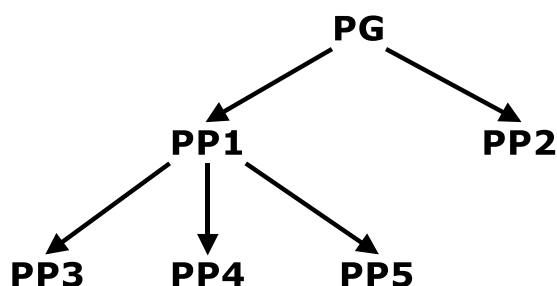
7.1.2 Procedury a programowanie strukturalne

W językach programowania oferujących mechanizm procedur zezwala się najczęściej, aby w ciele procedury znajdowały się wywołania innych procedur. Taki mechanizm pozwala na budowanie dużych programów metodą strukturalną:

1. Cały program jest zawarty w jednej niezbyt długiej *procedurze głównej*, która wywołuje *procedury podrzędne pierwszego poziomu*.
2. Procedury pierwszego poziomu wywołują *procedury drugiego poziomu*.
3. Procedury drugiego poziomu wywołują *procedury trzeciego poziomu*.
4. ...

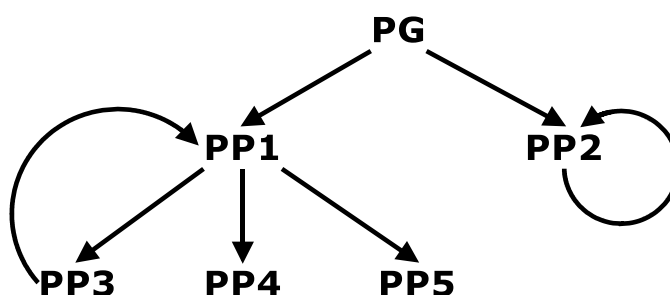
Liczba poziomów kolejnych procedur może być w zasadzie dowolna. Zalecenie praktyczne jest takie, aby deklaracja każdej z tych procedur mieściła się w całości na ekranie monitora, czyli aby mogła być „ogarnięta jednym rzutem oka”. Ta zasada pomaga programiście panować nad funkcjonalnością programu.

W najprostszym — ale też i najczęściej spotykanym przypadku — wzajemnie wywołujące się procedury tworzą hierarchię o strukturze drzewa, którego przykład widzimy na Rys. 7.1-1. Procedura główna PG wywołuje procedury podrzędne PP1 i PP2, a procedura PP1 wywołuje procedury PP3, PP4 i PP5.



Rys. 7.1-1 Drzewo procedur

Jednakże, może być też i tak, że procedura niższego poziomu wywołuje procedurę wyższego poziomu lub też samą siebie. Taką sytuację widzimy na Rys. 7.1-2.



Rys. 7.1-2 Graf procedur z rekurencją

Jeżeli w ciele procedury znajdzie się jej własne wywołanie, to w zależności od konstrukcji języka możemy mieć do czynienia w dwoma rodzajami sytuacji w chwili wywoływania tej procedury:

1. pojawi się komunikat błędu ‘procedura-nie zadeklarowana’,
2. procedura uruchomi kopię samej siebie.

Przypadek drugi — dziś typowy dla większości języków programowania — oznacza, że nastąpiło *rekurencyjne wywołanie procedury*. Taką rekurencję, gdzie PP2 wywołuje samą siebie, określam mianem *rekurencji prostej* dla odróżnienia jej od sytuacji, którą dalej nazwę *rekurencją wzajemną*, gdzie PP1 wywołuje PP3, a PP3 wywołuje PP1. Oczywiście przy rekurencji wzajemnej łańcuch wzajemnych wywołujących się procedur może być więcej niż dwuelementowy.

7.1.3 O procedurach denotacyjnie

Od pionierskich czasów, gdy zaczęto budować pierwsze języki programowania wyższego poziomu, mechanizm procedur pojawiał się w nich wszystkich, jednak szczegóły tego mechanizmu różniły się pomiędzy językami. I tak jest do dziś. Dla celów niniejszego wykładu przyjmuję więc pewien w miarę uniwersalny model, który — jak sądzę — powinien pozwolić czytelnikowi budować w przyszłości jego własne odmiany. Dobieram go też pod kątem budowania reguł dowodzenia poprawności programów.

O procedurach można powiedzieć, że są to instrukcje lub odpowiednio wyrażenia, którym nadano nazwy. Jednakże od instrukcji i wyrażeń różnią się kilkoma istotnymi cechami funkcjonalnymi, które pozwalają na ich wielokrotne wykorzystywanie w różnych kontekstach:

- można je zapisywać w środowisku procedur w celu wielokrotnego wykonania; każde takie wykonanie nazywamy *wywołaniem procedury* (ang. *procedure-call*),
- procedury mogą posługiwać się zmiennymi lokalnymi, które nie są dostępne poza wnętrzem procedury zwanym też *treścią procedury*; o tych zmiennych mówimy, że ich *obszarem widoczności* jest treść procedury,
- można im zadawać na wejściu listy wartości, które służą do inicjalizacji zmiennych lokalnych; te wartości nazywamy *aktualnymi parametrami wartościowymi* lub *aktualnymi parametrami wołanymi przez wartość* (ang. *called-by-value parameters*); różnym wywołaniom procedury można zadawać różne parametry wartościowe,
- można im zadawać na wejściu listę identyfikatorów zmiennych, których wartości początkowe będą przekazane do procedury i którym procedura przekaże na zakończenie odpowiadające im wartości końcowe; te identyfikatory nazywamy *aktualnymi parametrami referencyjnymi* lub *aktualnymi parametrami wołanymi przez referencję* (ang. *called-by-reference parameters*); różnym wywołaniom procedury można zadawać różne parametry referencyjne.

Od instrukcji i wyrażeń procedury różnią się dodatkowo jeszcze jedną charakterystyczną własnością: nie mają swoich odpowiedników na poziomie składni. We wszystkich znanych mi językach programowania procedury nie występują ani jako obiekty składniowe, ani nawet jako wyodrębnione pojęcia. Mówi się o treściach procedur, o ich deklaracjach i wywołaniach, ale o samych procedurach już nie. A dzieje się tak dlatego, że opisy języków programowania zwykle ograniczają się do składni⁷⁴.

W moim odczuciu mówienie o treściach, wywołaniach i deklaracjach czegoś, co nie zostało zdefiniowane, stoi w sprzeczności z matematycznym dobrym obyczajem i zwykle przekłada się na gorsze rozumienie mechanizmu procedur. A gorsze rozumienie to nieodmiennie źródło błędów. W denotacyjnym modelu języka **Lingua-2** wprowadzam więc pojęcie procedury jako niezależnego bytu po stronie denotacji. Ponieważ jednak procedury nie będą miały odpowiedników składniowych — na poziomie składni wystąpią jedynie deklaracje procedur i wywołania procedur — nie będę mówił o denotacjach procedur, ale o procedurach jako takich.

Dla wbudowania do języka mechanizmu procedur zdefiniuję trzy rodzaje obiektów wraz z odpowiadającymi im konstruktorami:

<i>procedura</i>	— funkcja modyfikująca składy (procedura imperatywna) lub wyliczająca kompozyt (procedura funkcyjna) ⁷⁵ ,
<i>denotacja deklaracji procedury</i>	— funkcja modyfikująca środowisko procedur przez powiązanie w środowisku nazwy procedury z nią samą
<i>denotacja wywołanie procedury</i>	— funkcja modyfikująca stan lub wyliczająca kompozyt przez uruchomienie procedury.

⁷⁴ Nie wspominając już o tym, że najczęściej nawet opis składni jest mocno nieprecyzyjny i niekompletny.

⁷⁵ W tym miejscu świadomie rezygnuję z procedur funkcyjnych mających tzw. „efekty uboczne” polegające na zmianie stanu obok wyliczenia kompozytu. W modelu denotacyjnym można by oczywiście takie procedury wprowadzić, to jednak oznaczałoby, że wyrażenia mają również takie efekty, bo — jak zobaczymy dalej — wywołanie procedury funkcyjnej będzie wyrażeniem. Takie rozwiązanie komplikowałoby jednak nie tylko sam model denotacyjny, ale przede wszystkim reguły budowania programów poprawnych.

Ponieważ w naszym modelu procedury zarówno reagują na sygnał błędu jak też mogą go generować założę, że są funkcjami na składach, a nie na wartościowaniach. Jak już wspominałem, pod względem typologicznym procedury dzielimy na dwa rodzaje:

1. *procedury imperatywne* odpowiadające instrukcjom z parametrami:

$$\text{pri} : \text{Prolmp} = \text{Parametry} \mapsto \text{Skład} \rightarrow \text{Skład}$$

2. *procedury funkcyjne* odpowiadające wyrażeniom z parametrami

$$\text{prf} : \text{ProFun} = \text{Parametry} \mapsto \text{Stan} \rightarrow \text{KompozytB}$$

Tak więc:

$$\text{pro} : \text{Procedura} = \text{Prolmp} \mid \text{ProFun}$$

Podążając za zwyczajem przyjętym w wielu językach programowania procedury imperatywne będę w skrócie nazywał *procedurami*, a procedury funkcyjne — *funkcjami*.

7.1.4 Dziedziny denotacyjne związane z procedurami

Zgodnie z zapowiedzią sformułowaną w rozdziale 4.3 algebra denotacji **Lingua-2** powstanie z algebry dla **Lingua-1** przez dodanie nowych nośników i nowych konstruktorów do jej algebry denotacji i odpowiednio składni. Poprzednie nośniki i konstruktory pozostaną niezmienione, natomiast pojawienie się nowych konstruktorów oznacza dodatkowo powiększenie dwóch nośników algebry składni odpowiadających wyrażeniom danologicznym i instrukcjom.

Jak zobaczymy dalej, deklaracje procedur imperatywnych przyjmują dwie listy parametrów formalnych, które nazywa się zwykle *formalnymi parametrami wartościowymi* i *formalnymi parametrami referencyjnymi*. Wywołania procedur przyjmują analogiczne dwie listy parametrów aktualnych: *aktualne parametry wartościowe* i *aktualne parametry referencyjne*. Dziedziny parametrów definiuję w sposób następujący:

$$\text{pfo} : \text{ParFor} = (\text{Identyfikator} \times \text{DenWyrTyp})^{c*} \quad (\text{parametry formalne deklaracji proc.})$$

$$\text{pak} : \text{ParAkt} = \text{Identyfikator}^{c*} \quad (\text{parametry aktualne wywołań proc.})$$

Parametry formalne obejmują wyrażenia typologiczne, gdyż w deklaracji procedury określamy typy jej przyszłych parametrów aktualnych. Procedury imperatywne modyfikują składy, a procedury funkcyjne wyliczają kompozyty:

$$\text{pri} : \text{Prolmp} = \text{ParAkt} \times \text{ParAkt} \mapsto \text{Skład} \rightarrow \text{Skład} \quad (\text{proc. imperatywne})$$

$$\text{prf} : \text{ProFun} = \text{ParAkt} \mapsto \text{Stan} \rightarrow \text{KompozytB} \quad (\text{proc. funkcyjne})$$

$$\text{pro} : \text{Procedura} = \text{Prolmp} \mid \text{ProFun} \quad (\text{procedury})$$

Procedury funkcyjne otrzymują przy ich wywoływaniu tylko jedną listę parametrów aktualnych, a mianowicie listę parametrów wartościowych. Kolejne dziedziny odpowiadają deklaracjom procedur obu typów:

$$\text{ddp} : \text{DenDekPri} = \text{Stan} \mapsto \text{Stan} \quad (\text{denotacje deklaracji procedur imperatywnych})$$

$$\text{ddf} : \text{DenDekPrf} = \text{Stan} \mapsto \text{Stan} \quad (\text{denotacje deklaracji procedur funkcyjnych})$$

Zauważmy, że w definicji dziedziny **Prolmp** nie mamy do czynienia z nielegalną rekurencją, gdyż procedury imperatywne nie przyjmują jako argumentów stanów wiążących w

środowiskach procedury, ale jedynie składy, czyli wartościowania⁷⁶. Dlatego właśnie wprowadziłem składy jako odrębny komponent stanu. Gdybyśmy przyjęli

$$\text{ProImp} = \text{ParAkt} \times \text{ParAkt} \mapsto \text{Stan} \rightarrow \text{Stan}$$

to wobec równań

$$\text{Stan} = (\text{ŚroTyp} \times \text{ŚroPro}) \times \text{Skład}$$

$$\text{ŚroPro} = \text{Identyfikator} \Rightarrow \text{ProImp} \mid \text{ProFun}$$

mielibyśmy do czynienia z nielegalnym układem stałopunktowym, gdyż operatory „ $\mapsto$ ” oraz „ $\rightarrow$ ” są nieciągłe (por. rozdział 2.7). Podobna sytuacja typologiczna występowała w języku Algol 60 (rozdział 4.1.), choć tam problem nie był związany z rekurencją, a polegał na tym, że procedury mogły brać same siebie jako parametry. Procedury, które mogą brać inne procedury — ale nie same siebie — jako parametry będą przedmiotem rozważań w rozdziale 7.6.

Dla zachowania prostoty naszego modelu, a w szczególności reguł budowania programów poprawnych, przyjąłem też, że parametry aktualne procedury, których zadaniem jest przekazanie do procedury kompozytów, są identyfikatorami, a nie denotacjami wyrażeń. Jak zobaczymy w dalszym ciągu, wyrażenia w **Lingua-2** obejmą wywołania procedur funkcyjnych, które z kolei mogą zawierać w sobie wywołania procedur imperatywnych. Gdyby aktualne parametry wartościowe mogły być wyrażeniami, to można by napisać procedurę, które rekurencyjnie wywołuje siebie przy wyliczaniu swojego parametru aktualnego. To denotacyjnie pewnie możliwe, ale dowodzenie własności takiego wywołania byłoby z pewnością dość złożone. Więcej na ten temat w rozdziale 7.5.1.

Po zdefiniowaniu dziedzin dla procedur należy przejść do definicji ich konstruktorów. W tym miejscu zakładam, że wszystkie konstruktory zdefiniowane w **Lingua-1** przenoszą się bez żadnych zmian do **Lingua-2**, natomiast dochodzą do nich nowe konstruktory związane z procedurami.

7.2 Komunikacja procedury imperatywnej z programem

W opisie mechanizmów związanych z procedurami będę używał pojęć⁷⁷ związanych z faktem, że procedura jest tworzona w czasie jej deklarowania, a wykonywana — w czasie wywołania. Będę więc mówił o środowiskach i wartościowaniach *czasu deklaracji procedury* i *czasu wywołania procedury*. Tradycyjnie denotację programu, z którego budujemy procedurę nazywam *treścią procedury*.

Jak już zapowiadałem, w **Lingua-2** nie dopuszczam w procedurach zmiennych globalnych. To nie matematyczna konieczność, ale świadoma decyzja budowniczego języka⁷⁸. Jej celem jest sprawienie, aby nagłówek procedury opisywał w sposób wyraźny i pełny mechanizm komunikacji procedury z goszczącym ją programem. Takie rozwiązanie może się wydawać nieco uciążliwe, jednakże zapewnia — moim zdaniem — lepsze panowanie programisty nad funkcjonalnością programu, a także upraszcza reguły budowania programów poprawnych.

⁷⁶ Takie właśnie rozwiązanie zostało zaproponowane w pracy [29], którą napisaliśmy wspólnie z Andrzejem Tarleckim w roku 1983.

⁷⁷ Jak kilka innych idei technicznych i tę zapożyczyłem od M. Gordona [45].

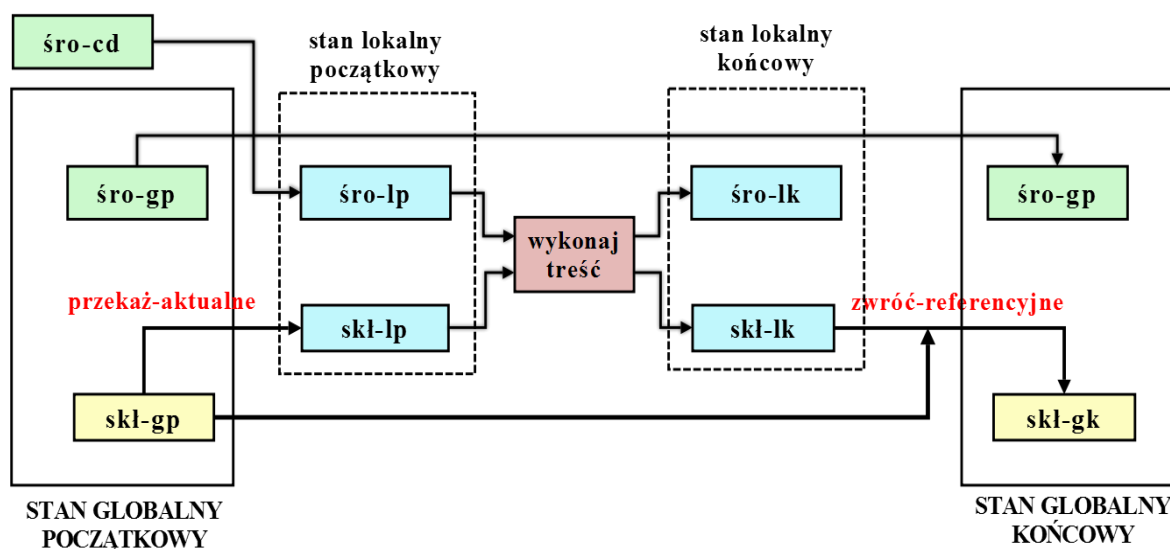
⁷⁸ Gdybyśmy chcieli wprowadzić do języka mechanizm zmiennych globalnych, trzeba by lokalny skład wykonania procedury tworzyć jako modyfikację składu globalnego.

7.2.1 Jak to działa?

Wykonanie instrukcji wywołania procedury można umownie podzielić na cztery etapy zilustrowane na Rys. 7.2-1. Szczegóły techniczne tej konstrukcji są opisane w rozdziale 7.3.

1. **Ocena stanu globalnego początkowego**, na który składają się:
 - a. *środowisko globalne początkowe* $\acute{s}ro-gp$,
 - b. *skład globalny początkowy* $sk\acute{l}-gp = (wrt-gp, b\acute{l}d-gp)$.

Jeżeli $b\acute{l}d-gp \neq OK$, to stan globalny początkowy staje się stanem globalnym końcowym wywołania. W przeciwnym przypadku jest tworzony stan lokalny początkowy.
2. **Tworzenie stanu lokalnego początkowego**, na który złożą się:
 - a. *środowisko lokalne początkowe* $\acute{s}ro-lp$, które powstaje ze środowiska $\acute{s}ro-cd$ czasu deklaracji procedury przez zagnieżdżenie w nim wywoływanej procedury; to zagnieżdżenie jest niezbędne dla umożliwienia wywołań rekurencyjnych (rozdział 7.3.3),
 - b. *skład lokalny początkowy* $sk\acute{l}-lp$, którego wartościowanie $wrt-lp$ obejmuje jedynie parametry formalne, którym przypisano wartości odpowiadających im parametrów aktualnych; dla wyliczenia tych wartości trzeba odwołać się do wartościowania globalnego początkowego $wrt-gp$ zawartego w składzie globalnym początkowym $sk\acute{l}-gp$.



Rys. 7.2-1 Wykonanie wywołania procedury

3. **Przetworzenie lokalnego stanu początkowego** przez wykonanie treści procedury. Jeżeli wykonanie treści procedury się zakończy, to w wyniku powstaje *stan lokalny końcowy*, na który składają się:
 - a. *środowisko lokalne końcowe* $\acute{s}ro-lk$,
 - b. *skład lokalny końcowy* $sk\acute{l}-lk = (wrt-lk, b\acute{l}d-lk)$.

Jeżeli $\text{błd-lk} \neq \text{OK}$, to stan globalny końcowy powstaje ze stanu globalnego początkowego przez załadowanie do niego komunikatu błędu błd-lk . Zauważmy, że w tym przypadku lokalne końcowe środowisko i skład zostają „porzucone”. W przeciwnym przypadku jest tworzony globalny stan końcowy.

4. **Tworzenie stanu globalnego końcowego**, na który złożą się:

- a. *środowisko globalne początkowe* śro-gp ; zauważmy, że środowisko lokalne końcowe zostaje „porzucone”,
- b. *skład globalny końcowy* skł-gk , który powstaje ze składu globalnego początkowego skł-gp przez „zwrócenie” do niego wartości referencyjnych parametrów formalnych (zapisanych w skł-lk) i przypisanie ich do odpowiadających im referencyjnych parametrów aktualnych.

Zauważmy, że treść procedury ma przez środowisko lokalne dostęp do wszystkich typów i procedur zapisanych w środowisku czasu deklaracji procedury. W tym środowisku może też zapisywać własne typy i procedury, które jednak mają charakter lokalny, tj. przestają być dostępne po zakończeniu wykonania wywołania. Po tym następuje powrót do środowiska początkowego czasu wywołania procedury. Na podkreślenie zasługuje też fakt, że treść procedury ma dostęp jedynie do tej części środowiska, która powstała przed deklaracją procedury, a nie przed jej wywołaniem.

Podobnie lokalny charakter ma tworzone na czas wywołania procedury wartościowanie lokalne, jednakże zapisane w nim wartości parametrów referencyjnych są na końcu przekazywane do wartościowania globalnego końcowego.

Opisany powyżej tryb wykonywania instrukcji wywołania procedury oznacza w szczególności, że:

1. jedyneymi zmiennymi widocznymi w ciele procedury są parametry formalne plus zmienne zadeklarowane lokalnie,
2. w ciele procedury są widoczne wszystkie globalne typy i procedury czasu deklaracji plus wszystkie podobne byty zadeklarowane lokalnie,
3. zmienne, typy i procedury zadeklarowane w ciele procedury, nie są widoczne na zewnątrz.

Taki wybór to nie matematyczna konieczność, ale kolejna decyzja o charakterze pragmatycznym mająca na celu względną prostotę modelu procedur, co przełoży się na prostotę reguł dowodzenia programów z procedurami, a także — jak sądzę — na lepsze panowanie nad funkcjonalnością programów przez użytkownika języka.

Na koniec jedna uwaga metodologiczna. Otóż z implementacyjnego punktu widzenia opisany mechanizm wymaga, aby stan globalny początkowy był przechowywany niezmienny (zapamiętywany) przez czas wykonywania procedury, by można było wrócić do niego na zakończenie. W konsekwencji fakt, że procedura może mieć wiele kolejnych wywołań rekurencyjnych oznacza, że każde wywołanie będzie „zapamiętywać” swoje stany początkowe. Ten mechanizm zwykle realizuje się przez budowanie stosu kolejnych stanów. To jednak sposób na zrealizowanie rekurencji w sposób iteracyjny (tj. bez rekurencji). W naszym przypadku uciekanie się do takiego modelu nie jest potrzebne, gdyż — jak zobaczymy w rozdziale 7.3.2 — rekurencję w **Lingua** możemy opisać za pomocą stałopunktowej rekurencji w **MetaSoft**.

7.2.2 Zgodność list parametrów

Przy wywoływaniu procedury imperatywnej jej parametry formalne otrzymują wartości parametrów aktualnych (tj. dane utypowane) i w ten sposób powstaje wartościowanie lokalne. Aby takie przekazanie wartości było możliwe, listy parametrów aktualnych w wywołaniu procedury muszą odpowiadać listom parametrów formalnych w deklaracji procedury zarówno co do liczności jak i typologicznie. Dodatkowo, wartości parametrów aktualnych muszą być określone. Dla opisanie tych warunków definiuję dwie funkcje sprawdzające, które w przyszłości zostaną wbudowane do definicji konstruktorów związanych z procedurami. Wśród ich argumentów znajdują się środowiska typów, które są niezbędne dla wyliczenia typów przypisanych parametrom formalnym.

statycznie-zgodne : ParFor x ParFor x ParAkt x ParAkt $\mapsto$ Błąd | {'OK'}

statycznie-zgodne.(pfo-w, pfo-r, pak-w, pak-r) =

niech (dla $n, m, k, p \geq 0$)

pfo-w = ((ide-fw.i, dwt-fw.i) | $i=1;k$) (parametry formalne wartościowe)

pfo-r = ((ide-fr.i, dwt-fr.i) | $i=1;n$) (parametry formalne referencyjne)

pak-w = (ide-aw.i | $i=1;p$) (parametry aktualne wartościowe)

pak-r = (ide-ar.i | $i=1;m$) (parametry aktualne referencyjne)

są-powtórzenia.[(ide-fr.i | $i=1;n$) ©

(ide-fw.i | $i=1;k$)] $\rightarrow$ 'powtórzenia-par-for'⁷⁹

są-powtórzenia.pak-r $\rightarrow$ 'powtórzenia-par-akt-ref'

$n \neq m$ lub $k \neq p$ $\rightarrow$ 'niezgodne-liczności-ref-for/akt'

prawda $\rightarrow$ 'OK'

Tak więc listy parametrów formalnych i aktualnych wywołania procedury uznajemy za statycznie zgodne, jeżeli:

1. żaden identyfikator na obu listach parametrów formalnych nie pojawia się dwukrotnie; parametry aktualne wartościowe nie podlegają takiemu ograniczeniu,
2. odpowiadające sobie listy parametrów formalnych i aktualnych są tej samej długości.

W tym przypadku mówimy o statycznej zgodności, gdyż można ją ustalić na etapie kompilacji programu, czyli przed etapem jego wykonania. Ale uwaga: „statycznie” nie oznacza „składniowo”. Tę analizę wykonujemy na etapie interpretacji programu, a więc po sprawdzeniu jego składniowej poprawności.

Kolejna funkcja zgodności odwołuje się do wartościowań i środowisk typów, ma więc charakter dynamiczny, gdyż jej wyliczenie jest możliwe dopiero na etapie wykonywania programu. Tu również parametry formalne są porównywane z aktualnymi.

dynamicznie-zgodne : ParFor x ParFor x ParAkt x ParAkt $\mapsto$

ŚroTyp x Wart $\mapsto$ Błąd | {'OK'}

⁷⁹ Funkcja **są-powtórzenia** (patrz rozdział 2.1.4) jest zdefiniowana na krotkach. Stąd jej argumentem w tej definicji jest konkatenacja „©” ciągu parametrów formalnych referencyjnych z ciągiem parametrów formalnych wartościowych.

dynamicznie-zgodne.(pfo-w, pfo-r, pak-w, pak-r).(śrt, wrt) =

niech

komunikat = statycznie-zgodne.(pfo-w, pfo-r, pak-w, pak-r)

komunikat : Błąd → komunikat

niech (dla $n, m, k, p \geq 0$)

pfo-w = ((ide-fw.i, dwt-fw.i) | $i=1;k$) (parametry formalne wartościowe)

pfo-r = ((ide-fr.i, dwt-fr.i) | $i=1;n$) (parametry formalne referencyjne)

pak-w = (ide-aw.i | $i=1;p$) (parametry aktualne wartościowe)

pak-r = (ide-ar.i | $i=1;m$) (parametry aktualne referencyjne)

badanie określoności aktualnych parametrów wartościowych

$(\exists i) \text{ wrt.}(\text{ide-aw.i}) = ? \rightarrow \text{'parametr-wartościowy-niezadeklarowany'}$

niech

$((\text{dan-aw.i}, \text{kor-aw.i}), \text{tra-aw.i}) = \text{wrt.ide-aw.i}$ dla $i = 1;k$

$(\exists i) \text{ dan-aw.i} = \Omega \rightarrow \text{'parametr-wartościowy-niezainicjalizowany'}$

badanie określoności aktualnych parametrów referencyjnych

$(\exists i) \text{ wrt.}(\text{ide-ar.i}) = ? \rightarrow \text{'parametr-referencyjny-niezadeklarowany'}$

niech

$((\text{dan-ar.i}, \text{kor-ar.i}), \text{tra-ar.i}) = \text{wrt.ide-ar.i}$ dla $i = 1;m$

$(\exists i) \text{ dan-ar.i} = \Omega \rightarrow \text{'parametr-referencyjny-niezainicjalizowany'}$

wyliczenie typów formalnych parametrów wartościowych

niech

sta = ((śrt, []), (wrt, 'OK')) (wyjaśnienie w komentarzach)

typ-fw.i = dwt-fw.i.sta dla $i = 1;k$ (typy par. form. wartościowych)

typ-fr.i = dwt-fr.i.sta dla $i = 1;n$ (typy par. form. referencyjnych)

$(\exists i) \text{ typ-fw.i} : \text{Błąd} \rightarrow \text{'błąd-wyliczenia-typu-parametru-fw'}$

$(\exists i) \text{ typ-fr.i} : \text{Błąd} \rightarrow \text{'błąd-wyliczenia-typu-parametru-fr'}$

niech

$(\text{kor-fw.i}, \text{tra-fw.i}) = \text{typ-fw.i}$ dla $i = 1;k$

$(\text{kor-fr.i}, \text{tra-fr.i}) = \text{typ-fr.i}$ dla $i = 1;n$

$(\exists i) \text{ kor-fw.i} \neq \text{kor-aw.i} \rightarrow \text{'niezgodność-korpusów-war-for/akt'}$

$(\exists i) \text{ kor-fr.i} \neq \text{kor-ar.i} \rightarrow \text{'niezgodność-korpusów-ref-for/akt'}$

$(\exists i) (\text{tra-fw.i}).((\text{dan-aw.i}, \text{kor-aw.i}) \neq (\text{pp}, (\text{'boolowska'})) \rightarrow \text{'niesp.-jarzmo-war'}$

$(\exists i) (\text{tra-fr.i}).((\text{dan-ar.i}, \text{kor-ar.i}) \neq (\text{pp}, (\text{'boolowska'})) \rightarrow \text{'niesp.-jarzmo-ref'}$

prawda → 'OK'

Tak więc listy parametrów formalnych i aktualnych wywołania procedury uznajemy za dynamicznie zgodne, jeżeli:

1. są statycznie zgodne,
2. wszystkie parametry aktualne zostały zadeklarowane i zainicjalizowane,
3. wszystkie wyrażenia typologiczne określające typy parametrów formalnych obu rodzajów mają wartości nie będące komunikatami błędu,
4. korpusy odpowiadających sobie referencyjnych parametrów formalnych i aktualnych zarówno wartościowych, jak i referencyjnych, są identyczne; typ parametru formalnego jest określony wyrażeniem typologicznym zawartym w deklaracji procedury, a aktualnego — w wartościowaniu bieżącym dla wywołania procedury,
5. kompozyty niesione przez parametry aktualne spełniają jarzma odpowiadających im parametrów formalnych; zauważmy, że jarzma związane z parametrami aktualnymi w ogóle nie są brane pod uwagę.

Dla wyliczenia typów przypisanych parametrom formalnym trzeba było zastosować pewien wybieg techniczny. Otóż te typy są zdefiniowane na poziomie składni przez wyrażenia typologiczne, a więc na poziomie denotacji — przez denotacje takich wyrażen. Jednak aby z denotacji wyliczyć typ, trzeba ją zastosować do stanu, a funkcja

dynamicznie-zgodne.(pfo-w, pfo-r, pak-w, pak-r)

otrzymuje jako argument jedynie parę (śrt, wrt). Tworzy się zatem „tymczasowy” stan

((śrt, []), (wrt, 'OK'))

gdzie [] jest pustym środowiskiem procedur. To środowisko może być zresztą dowolne, bo denotacje wyrażen typologicznych do niego nie sięgają.

Zauważmy, że w definicji obu funkcji dopuszczamy wartości zerowe każdej z liczb n , m , k i p , co odpowiada pustym listom parametrów.

7.2.3 Przekazywanie parametrów aktualnych do procedury

Ta funkcja jest aktywowana w momencie wywołania procedury i tworzy wartościowanie lokalne początkowe (rozdział 7.1.1). Jedyne identyfikatory wiązane w tym wartościowaniu to parametry formalne obu rodzajów, a ich wartości początkowe to wartości odpowiadających im parametrów aktualnych.

przekaż-aktualne : ParFor x ParFor x ParAkt x ParAkt $\mapsto$

ŚroTyp x Wart $\mapsto$ Wart | Błąd

przekaż-aktualne.(pfo-w, pfo-r, pak-w, pak-r).(śrt, wrt) =

niech

komunikat = dynamicznie-zgodne.(pfo-w, pfo-r, pak-w, pak-r).(śrt, wrt)

komunikat $\neq$ 'OK' $\rightarrow$ komunikat

niech (dla n , $k \geq 0$)

((ide-fw.i, dwt-fw.i) | $i=1;k$) = pfo-w (parametry formalne wartościowe)

((ide-fr.i, dwt-fr.i) | $i=1;n$) = pfo-r (parametry formalne referencyjne)

(ide-aw.i | $i=1;k$) = pak-w (parametry aktualne wartościowe)

$(ide-ar.i \mid i=1;n) = pak-r$ (parametry aktualne referencyjne)
 $war-w.i = wrt.(ide-aw.i)$ dla $i=1;k$ (wartości akt. parametrów wartościowych)
 $war-r.i = wrt.(ide-ar.i)$ dla $i=1;n$ (wartości akt. parametrów referencyjnych)

tworzenie lokalnego wartościowania początkowego

$wrt-w = [ide-fw.i/war-w.i \mid i=1;k]$ (wartościowanie lokalne pocz. par. wart.)
 $wrt-r = [ide-fr.i/war-r.i \mid i=1;n]$ (wartościowanie lokalne pocz. par. ref.)
 $wrt-lp = wrt-w \blacklozenge wrt-r$ (wartościowanie lokalne początkowe)⁸⁰

prawda $\rightarrow wrt-lp$

Operator przekazywania parametrów w pierwszym rzędzie przeprowadza opisaną wcześniej analizę zgodności parametrów, a następnie tworzy wartościowanie lokalne początkowe, w którym później nastąpi wykonanie treści procedury:

1. formalnym parametrom wartościowym przypisuje się wartości aktualnych parametrów wartościowych; określoność tych wartości i zgodność ich typów została sprawdzona przez funkcję poprawności dynamicznie-zgodne.
2. formalnym parametrom referencyjnym przypisuje się wartości aktualnych parametrów referencyjnych; określoność tych wartości i zgodność ich typów została sprawdzona przez funkcję poprawności dynamicznie-zgodne,

Podobnie jak w poprzednich definicjach dopuściłem puste listy parametrów.

Opisując mechanizm tworzenia wartościowania lokalnego początkowego wykluczyłem obecność w nim zmiennych globalnych, które byłyby widoczne zarówno wewnątrz jak i na zewnątrz procedury. Jedynym kanałem komunikacji, którym procedura może cokolwiek przekazać „na zewnątrz” swojego wywołania stanowią więc aktualne i formalne parametry referencyjne, które przekazują sobie wartości według poniższego schematu:

$pfo-w := wartości\ pak-w$
 $pfo-r := wartości\ pak-r$
 wykonanie treści procedury
 $pak-r := wartości\ pfo-r$

7.2.4 Zwracanie parametrów referencyjnych do programu

O ile formalne parametry wartościowe grają w ciele procedury rolę zmiennych lokalnych, a więc widocznych jedynie wewnątrz treści procedury, o tyle formalne parametry referencyjne grają rolę zmiennych globalnych, których wartości są modyfikowane przez procedurę.

$zwróc-referencyjne : ForParRef \times AktParRef \mapsto \acute{S}roTyp \times Wart \times Wart$
 $\mapsto Wart \mid Błąd$
 $zwróc-referencyjne.(pfo-r, pak-r).(śrt, wrt-lk, wrt-gp) =$
niech

⁸⁰ Wartościowanie lokalne powstaje jako przesłonięcie lokalnego wartościowania referencyjnego lokalnym wartościowaniem wartościowym. Ponieważ odpowiadające im zbiory identyfikatorów są rozłączne odpowiada to sklejeniu tych funkcji. Operacja przesłonięcia $\blacklozenge$ została zdefiniowana w rozdziale 2.1.3.

komunikat = dynamicznie-zgodne.(), pfo-r, (), pak-r).(śrt, wrt-gp)⁸¹

komunikat $\neq$ 'OK' $\rightarrow$ komunikat

niech

(ide-ar.i | i=1;n) = pak-r (aktualne parametry referencyjne)

((ide-fr.i, typ-fr.i) | i=1;k) = pfo-r (formalne parametry referencyjne)

($\exists i$) wrt-lk.(ide-fr.i) = ? $\rightarrow$ 'nieokreślona-wartość-parametru-referencyjnego'

niech

war-fr.i = wrt-lk.(ide-fr.i) dla i=1;n (wartości końcowe form. par. ref.)

wrt-gk = wrt-gp[ide-ar.i/war-fr.i | i=1;n] (wartościowanie globalne końcowe)

prawda

$\rightarrow$ wrt-gk

Po wykonaniu treści procedury, wartości formalnych parametrów referencyjnych są przekazywane odpowiadającym im aktualnym parametrom referencyjnym.

Komunikacja pomiędzy wnętrzem i zewnątrz procedury wygląda więc tak, że w chwili wywołania procedury formalne parametry obu typów — a więc identyfikatory występujące w ciele procedury — otrzymują wartości parametrów aktualnych, a w chwili zakończenia wykonania procedury formalne parametry referencyjne oddają te wartości swoim odpowiednikom aktualnym. Zgodności list parametrów aktualnych z formalnymi nie musimy sprawdzać, bo — jak zobaczymy — zostanie to dokonane przy uruchamianiu wywołania procedury.

Gdybyśmy ten sposób komunikacji chcieli opisać w terminach implementacyjnych, to — jak już pisałem wcześniej — odpowiadałby on dwóm symbolicznym instrukcjom przypisania. Przed wykonaniem treści procedury

pfo-r := wartości pak-r

a po jej wykonaniu

pak-r := wartości pfo-r.

Oznaczałoby to więc kopiowanie wartości parametrów aktualnych do obszaru pamięci zarezerwowanego dla wykonania procedury. Jeżeli wartością parametru jest obiekt nie zajmujący w pamięci dużo miejsca, na przykład liczba, to takie rozwiązanie jest akceptowalne. Jeżeli jednak jest to duży obiekt, np. baza danych, to może być ono wręcz niewykonalne ze względu na niedostatek pamięci, nie mówiąc już o czasie potrzebnym na skopiowanie dużej bazy.

Dla rozwiązania tego problemu w implementacji większości języków na wejściu wykonania procedury parametr formalny otrzymuje nie wartość parametru aktualnego, ale „dostęp do tej wartości”, a więc adres (referencję), pod którym ta wartość jest zapisana na dysku. Pod względem funkcjonalnym (wynikowym) to rozwiązanie jest równoważne naszemu, gdybyśmy jednak chcieli opisać je literalnie w modelu denotacyjnym, należałoby wprowadzić pojęcie adresu i wiązać identyfikatory z adresami, a adresy z wartościami. Takie rozwiązanie przyjął właśnie M. Gordon [45]. Wybór pomiędzy oboma rozwiązaniami zależy od adresata naszego modelu — czy jest to użytkownik języka, czy też twórca implementacji. Zgodnie z przyjętą w niniejszej książce filozofią, jeżeli opisujemy język pod kątem użytkownika, to opisujemy jego

⁸¹ Przez „()” oznaczyłem puste krotki parametrów. Dzięki temu mogłem zastosować do dwóch list parametrów funkcję o czterech argumentach. Zauważmy, że w naszej definicji w zasadzie nie musimy sprawdzać zgodności parametrów, ponieważ zostaną one sprawdzone przy przekazywaniu parametrów do procedury (rozdział 7.2.3). Nie można ich jednak usunąć z definicji naszej funkcji bez naruszenia jej poprawności.

funkcjonalność widzianą przez użytkownika, a niekoniecznie realizowaną implementacyjnie. Z tego właśnie powodu nie wprowadziłem do modelu pojęcia adresu.

7.3 Procedury imperatywne z rekurencją pojedynczą

Jak już zapowiadałem, w denotacyjnym modelu **Lingua-2** wprowadzam pojęcie procedury jako niezależnego obiektu. W tym celu posługuję się konstrukcją opisaną w pracy [29], którą opublikowaliśmy wspólnie z Andrzejem Tarleckim w roku 1983. Przypominam, że w przypadku procedur nie mówimy o denotacji procedury, gdyż procedury nie mają swoich odpowiedników po stronie składni języka, występują więc one jedynie po stronie denotacji. Odpowiedniki składniowe będą natomiast miały deklaracje procedur i wywołania procedur.

7.3.1 Konstruktory parametrów

Ponieważ procedury będą budowane przez konstruktory przyjmujące parametry jako swoje argumenty, należy rozpocząć od zdefiniowania konstruktorów parametrów. Parametry aktualne są krotkami identyfikatorów (być może pustymi) można je więc budować za pomocą trzech konstruktorów:

twórz-pusty-akt : $\mapsto$ ParAkt

twórz-pusty-akt.() = ()

twórz-par-akt : Identyfikator $\mapsto$ ParAkt

twórz-par-akt.ide = (ide)

sekwencja-par-akt : ParAkt $\times$ ParAkt $\mapsto$ ParAkt

sekwencja-par-akt.(pak-1, pak-2) = pak-1 $\odot$ pak-2

W analogiczny sposób definiujemy konstruktory parametrów formalnych:

twórz-pusty-for : $\mapsto$ ParFor

twórz-pusty-for.() = ()

twórz-par-for : Identyfikator $\times$ DenWyrTyp $\mapsto$ ParFor

twórz-par-for.(ide, dwt) = (ide, dwt)

sekwencja-par-for : ParFor $\times$ ParFor $\mapsto$ ParFor

sekwencja-par-for.(pfo-1, pfo-2) = pfo-1 $\odot$ pfo-2

W tym miejscu jest potrzebny pewien komentarz. Otóż konstruktor **twórz-par-for** robi wrażenie funkcji identycznościowej. W rzeczywistości jednak jest to funkcja, która z dwóch argumentów tworzy dwuelementową krotkę. Gdybyśmy krotki zapisywali w nawiasach kwadratowych, a nie zwykłych, to sytuacja byłaby jaśniejsza:

$$\text{twórz-par-for}(\text{ide}, \text{dwt}) = [\text{ide}, \text{dwt}]$$

Pozostanę jednak przy obecnym zapisie, gdyż został on już dawno utrwalony w matematyce.

7.3.2 Konstruktor procedury

Dla zdefiniowania konstruktora procedur i konstruktora denotacji deklaracji procedur wprowadzam pomocniczą dziedzinę *składowe procedury imperatywnej*

$$\text{spi} : \text{SkładowePri} = \text{Identyfikator} \times \text{ParFor} \times \text{ParFor} \times \text{DenPro}$$

Składowymi procedury są więc: identyfikator, dwie listy parametrów formalnych (wartościowe i referencyjne) oraz denotacja programu (treść procedury). Konstruktor procedury przyjmuje jako argumenty składowe procedury plus środowisko⁸², a zwraca procedurę jest więc typu:

$$\text{twórz-pro-imp} : \text{SkładowePri} \times \text{Środ} \mapsto \text{ProImp}$$

lub w postaci rozwiniętej:

$$\begin{aligned} \text{twórz-pro-imp} : ((\text{Identyfikator} \times \text{ParFor} \times \text{ParFor} \times \text{DenPro}) \times \text{Środ}) \mapsto \\ \text{ParAkt} \times \text{ParAkt} \mapsto \text{Skład} \rightarrow \text{Skład} \end{aligned}$$

Występujące w tej definicji środowisko jest środowiskiem czasu deklaracji (oznaczamy je przez *śro-cd*), bo procedura jest „tworzona” w chwili jej deklarowania⁸³. Wartością naszego konstruktora jest funkcja:

$$P = \text{twórz-pro-imp}((\text{ide}, \text{pfo-w}, \text{pfo-r}, \text{dpr}), \text{śro-cd})$$

która jest procedurą, a więc funkcją typu:

$$P : \text{ParAkt} \times \text{ParAkt} \mapsto \text{Skład} \rightarrow \text{Skład}$$

Tę funkcję definiujemy jako rozwiązanie równania stałopunktowego, w którego zapisie przyjąłem następujące oznaczenia:

$(\text{śrt-cd}, \text{śrp-cd}) = \text{śro-cd}$	— środowisko czasu deklaracji
skł-gp	— skład globalny początkowy
$\text{par} = (\text{pfo-w}, \text{pfo-r}, \text{pak-w}, \text{pak-r})$	— krotka parametrów związanych z procedurą
ide	— nazwa deklarowanej procedury
dpr	— denotacja programu będącego treścią procedury

Poniżej równanie definiujące procedurę *P*. Odwołuję się ono zarówno do wymienionych wyżej argumentów funkcji **twórz-pro-imp** jak i do argumentów definiowanej funkcji *P*. Skład *skł-gp* jest oczywiście składem z czasu wywołania procedury.

$$P(\text{pak-w}, \text{pak-r}).\text{skł-gp} =$$

⁸² Dlaczego środowiska nie zaliczyłem do składowych procedury okaże się przy budowaniu modelu dla wieloprocedur (rozdz.7.4).

⁸³ Zauważmy, że gdyby procedura miała być wykonywana w środowisku czasu wywołania, to musiałby być funkcją typu $P:\text{ParAkt} \times \text{ParFor} \mapsto \text{Środ} \rightarrow \text{Skład}$, co oznaczałoby, że brałaby samą siebie jako argument, a to, jak już wiemy, prowadzi do konstrukcji niedefiniowalnej na gruncie teorii mnogości.

jest-błąd.skł-gp $\rightarrow$ skł-gp

niech

(wrt-gp, 'OK') = skł-gp

wrt-lp = przekaz-aktualne.par.(śrt-cd, wrt-gp) (wartościowanie lok. początkowe)

wrt-lp : Błąd $\rightarrow$ wrt-lp

niech

skł-lp = (wrt-lp, 'OK') (skład lokalny początkowy)

śro-lp = (śrt-cd, śrp-cd[ide/P]) (środowisko lok. początkowe)

sta-lp = (śro-cd, skł-lp) (stan lok. początkowy)

dpr.sta-lp = ? $\rightarrow$?

niech (wykonanie treści procedury)

sta-lk = dpr.sta-lp (stan lokalny końcowy)

(śro-lk, (wrt-lk, błąd)) = sta-lk

błąd $\neq$ 'OK' $\rightarrow$ (wrt-gp, błąd) (*)

niech

wrt-gk = zwróć-referencyjne.(pfo-r, pak-r).(wrt-lk, wrt-gp)

wrt-gk : Błąd $\rightarrow$ (wrt-gp, wrt-gk) (**) (tu wrt-gk jest błędem)

prawda $\rightarrow$ (wrt-gk, 'OK')

W pierwszym kroku następuje sprawdzenie, czy skład globalny początkowy nie niesie błędu, a jeżeli tak jest, to staje się on składem globalnym końcowym.

W przeciwnym przypadku jest tworzone *wartościowanie lokalne początkowe* wrt-lp, w którym rozpocznie się wykonywanie treści procedury (por. Rys. 7.2-1). Powstaje ono z *wartościowania globalnego początkowego* wrt-gp przez przekazanie parametrom formalnym wartości odpowiadających im parametrów aktualnych. Ponieważ operator *przekaz-aktualne* bada poprawność list parametrów (rozdział 7.2.3), może też wygenerować błąd.

Jeżeli komunikat błędu się nie pojawi, to powstaje *skład lokalny początkowy* skł-lp z komunikatem 'OK' oraz jest tworzone *środowisko lokalne początkowe* śro-lp przez zagnieżdżenie (właśnie definiowanej) procedury P w środowisku czasu deklaracji. To zagnieżdżenie powoduje, że równanie definiujące P ma charakter stałopunktowy.

Zauważmy, że gdybyśmy nie umieścili definicyjnie procedury P w środowisku czasu deklaracji, to próba wywołania P w czasie wykonywania jej treści spowodowałaby komunikat *procedura-niezadeklarowana*, gdyż w środowisku czasu deklarowania procedury nie jest ona zadeklarowana. Należy podkreślić, że opisany tu mechanizm rekurencji nie obejmuje sytuacji, w której dwie lub więcej procedur wywołuje się wzajemnie. Tym przypadkiem zajmę się w rozdziale 7.4.

Środowisko lokalne początkowe wraz ze składem lokalnym początkowym tworzą *stan lokalny początkowy* sta-lp.

W kolejnym kroku treść procedury reprezentowanej przez dpr jest wykonywane w stanie lokalnym początkowym i jeżeli to wykonanie się zakończy, to powstaje *stan lokalny końcowy* sta-lk.

Jeżeli ten stan niesie błąd, to składem końcowym staje się skład złożony z globalnego wartościowania początkowego i tego błędu.

W przeciwnym przypadku z tego stanu jest wyodrębniane *wartościowanie lokalne końcowe* wrt-lk. Zapisane w nim wartości referencyjnych parametrów formalnych zostają przekazane do wartościowania globalnego początkowego wrt-gp i tam przypisane do referencyjnych parametrów aktualnych. W ten sposób powstaje *wartościowanie globalne końcowe* wrt-gk, które staje się składową *składu globalnego końcowego* (wrt-gk, 'OK').

Zauważmy, że jeżeli w ciele deklarowanej procedury nie występuje jej wywołanie, to efekt wykonania treści procedury w stanie o zmodyfikowanym środowisku czasu deklaracji $\text{śrp-cd}[\text{ide}/P]$ jest identyczny z wykonaniem tej treści w stanie niezmodyfikowanym śrp-cd . Definicja procedury z rekurencją pojedynczą obejmuje więc przypadek procedury bez rekurencji.

Procedura P jest funkcją, która po otrzymaniu parametrów aktualnych zwraca funkcję ze składu w skład. Zauważmy, że ani środowisko śro-cd ani identyfikator ide ani też denotacja programu dpr nie są argumentami procedury. One zostały jedynie użyte do jej zbudowania i zostały w niej niejako „zaszyte”, gdyż są w rzeczywistości parametrami równania definiującego procedurę. Jak zobaczymy dalej, zostaną one wskazane przez deklarację procedury.

Warto też zauważyć, że w wykonaniu naszej procedury mamy do czynienia z nietrywialną obsługą błędów, a więc inną niż tylko przekazanie błędów do dalszych partii programu. W klauzulach (*) oraz (**) pojawienie się błędów nie tylko przerywa bieg programu, ale też przywraca wartościowanie początkowe czyniąc go końcowym. To oczywiście tylko jedna z możliwych strategii przy obsłudze błędów.

Na koniec uwaga terminologiczna dotycząca rozróżnienia pomiędzy aktualnymi parametrami wartościowymi i referencyjnymi. W rzeczywistości te dwa rodzaje parametrów nie różnią się niczym pod względem denotacyjnym, gdyż w obu przypadkach są ciągami identyfikatorów. Różnica pomiędzy nimi leży w sposobach ich użycia w wywołaniu procedury. Stąd poprawniejsze byłoby używanie historycznych pojęć *parametrów wołanych przez wartość* i *parametrów wołanych przez referencję*, choć językowo nie brzmią one zbyt dobrze.

7.3.3 Instrukcja wywołania procedury imperatywnej

Wywołanie procedury to pobranie jej ze środowiska (gdzie została zadeklarowana, o czym za chwilę), a następnie „załadowanie” do niej parametrów aktualnych, w wyniku czego powstaje denotacja instrukcji. Tak więc:

$\text{wołaj-pro-imp} : \text{Identyfikator} \times \text{ParAkt} \times \text{ParAkt} \mapsto \text{DenIns}$

a zatem:

$\text{wołaj-pro-imp}(\text{ide}, \text{pak-w}, \text{pak-r}) : \text{Stan} \rightarrow \text{Stan}$

Tę denotację instrukcji definiujemy w sposób następujący:

$\text{wołaj-pro-imp}(\text{ide}, \text{pak-w}, \text{pak-r}).\text{sta-pw} =$

$\text{jest-błąd}.\text{sta-pw} \rightarrow \text{sta-pw}$

niech

$(\text{śro-pw}, \text{skł-pw}) = \text{sta-pw}$ (początkowy stan wywołania)

$(\text{śrt-pw}, \text{śrp-pw}) = \text{śro-pw}$ (początkowe środowisko wywołania)

$(\text{wrt-pw}, \text{'OK'}) = \text{skł-pw}$ (początkowy skład wywołania)

śrp-pw.ide = ? $\rightarrow$ sta-pw $\leftarrow$ 'procedura-nie zadeklarowana'

śrp-pw.ide : ProFun $\rightarrow$ sta-pw $\leftarrow$ 'procedura-nieimperatywna'

niech

pri = śro-pw.ide (wywoływana procedura imperatywna)

pri.(pak-w, pak-r).skł-pw = ? $\rightarrow$?

niech

skł-kw = pri.(pak-w, pak-r).skł-pw (skład końcowy wywołania)

(wrt-kw, błd) = skł-kw

błd $\neq$ 'OK' $\rightarrow$ sta-pw $\leftarrow$ błd

prawda $\rightarrow$ (śro-pw, skł-kw)

Jeżeli stan nie niesie komunikatu błędu oraz identyfikatorowi procedury ide jest w środowisku przypisana procedura imperatywna pri to aplikujemy tę procedurę do parametrów aktualnych, w wyniku czego otrzymujemy funkcję częściową modyfikującą składy:

pro.(pak-r, pak-w) : Skład $\rightarrow$ Skład

Tę funkcję aplikujemy do składu początkowego wywołania skł-pw. Zwróćmy uwagę, że ponieważ w procedurze jest zaszyte środowisko czasu deklaracji, to program będący treścią procedury wykonujemy w stanie (śro-cd, skł-pw) gdzie:

śro-cd — środowisko czasu deklaracji

skł-pw — skład początkowy wywołania

Jeżeli skład wynikowy nie jest określony, to nie jest również określony stan końcowy instrukcji wywołania. Jeżeli wykonanie treści procedury generuje komunikat błędu, to ten komunikat jest wprowadzany do stanu początkowego wywołania. W przeciwnym przypadku skład końcowy wywołania skł-kw staje się częścią stanu końcowego wywołania (śro-pw, skł-kw). Środowisko wywołania pozostaje bez zmian, co oznacza, że wszystkie zadeklarowane w ciele procedury obiekty (procedury i typy) przestają być widoczne.

Zauważmy, że w definicji tego konstruktora nie występuje w postaci wyraźnej badanie zgodności parametrów aktualnych z formalnymi, ani przekazywanie parametrów, gdyż te operacje zostały „zaszyte” w procedurze (rozdz.7.3.1). Występuje natomiast badanie typu procedury z względu na to, że do języka zostaną wprowadzone również procedury funkcyjne (rozdział 7.5).

7.3.4 Deklaracja procedury imperatywnej

Deklaracja procedury jest konstruktorem typu

deklaruj-pro-imp : SkładowePri $\mapsto$ DenDekPri

czyli typu

deklaruj-pro-imp : SkładowePri $\mapsto$ Stan $\mapsto$ Stan

zdefiniowanym w sposób następujący:

deklaruj-pro-imp.spi.sta =

jest-błąd.sta $\rightarrow$ sta

niech

$(ide, pfo-w, pfo-r, dpr) = spi$
 $(\acute{s}ro, (wrt, bld)) = sta$
 $(\acute{s}rt, \acute{s}rp) = \acute{s}ro$
 $ide : zadeklarowane.sta \rightarrow sta \leftarrow 'nazwa-' ide '-zaj\acute{e}ta'$

niech

komunikat = brak-powtórzeń.(pfo-r © pfo-w)
komunikat $\neq$ 'OK' $\rightarrow sta \leftarrow komunikat$

niech

$P = \text{twórz-pro-imp.}(spi, \acute{s}ro)$ (proc. jest tworzona w środowisku czasu deklaracji)
 $\acute{s}ro-k = (\acute{s}rt, \acute{s}rp[ide/P])$ (środowisko końcowe)

prawda $\rightarrow (\acute{s}ro-k, sto, 'OK')$

Tak więc deklaracja procedury tworzy procedurę, a następnie przypisuje ją identyfikatorowi *ide* w aktualnym środowisku deklaracji. Ten właśnie identyfikator i to środowisko otrzymuje jako swoje argumenty konstruktor procedury (por. rozdział 0).

7.4 Procedury imperatywne z rekurencją wzajemną

7.4.1 Rekurencja wzajemna

Jak już wyjaśniałem, opisana do tej pory rekurencja nie obejmuje przypadku, gdy procedura *P* wywołuje procedurę *Q*, która z kolei wywołuje *P*. Oczywiście na poziomie składni nie możemy takiej sytuacji wykluczyć, jednakże na poziomie denotacji (implementacji) — przy semantyce procedur zdefiniowanej w rozdziale 7.3 — spowoduje ona wygenerowanie komunikatu *procedura-niezadeklarowana*. Gdyby bowiem deklaracja *P* poprzedzała deklarację *Q*, to wywołanie *Q* w środowisku czasu deklaracji *P* nie „znalazłoby” w tym środowisku procedury *Q* i podobnie, gdyby pierwsze było *Q*.

Aby rozwiązać ten problem, *P* i *Q* trzeba zdefiniować wspólnym układem równań stałopunktowych postaci:

$$P = F(P, Q)$$

$$Q = G(P, Q)$$

i oczywiście analogicznie przy większej liczbie wzajemnie rekurencyjnych procedur.

Na poziomie algebry denotacji oznacza to wprowadzenie konstruktorów pozwalających na tworzenie krotek wzajemnie wywołujących się procedur. Takie krotki będę nazywał *wieloprocedurami*. Dla zdefiniowania konstruktora wieloprocedury wprowadzam trzy nowe dziedziny:

$$swp : \text{SkładoweWiePro} = \text{SkładowePri}^{c+} \quad (\text{składowe wieloprocedur})$$

$$wpr : \text{WiePri} = \text{ProImp}^{c+} \quad (\text{wieloprocedury})$$

$$ddw : \text{DenDekWiePri} = \text{Stan} \mapsto \text{Stan} \quad (\text{denotacja deklaracji wieloprocedury})$$

Składowe wieloprocedury tworzą niepustą krotkę składowych procedur pojedynczych, a wieloprocedura jest niepustą krotką procedur. Dziedzina denotacji deklaracji wieloprocedur jest identyczna z analogiczną dziedziną dla procedur pojedynczych, jednakże ich podzbiory osiągalne będą różne.

7.4.2 Konstruktor wieloprocedury

Konstruktor wieloprocedury jest funkcją typu:

$$\text{twórz-wiepro-imp} : \text{SkładoweWiePro} \times \text{Środ} \mapsto \text{WiePri}$$

zdefiniowaną w sposób analogiczny do konstruktora pojedynczej procedury⁸⁴. Niech

$$\text{swp} = ((\text{ide.i}, \text{pfo-w.i}, \text{pfo-r.i}, \text{dpr.i}) \mid i = 1;n) \quad (\text{składowe wieloprocedury})$$

Wtedy

$$\text{twórz-wiepro-imp}(\text{swp}, \text{śro-cd}) = (P.i \mid i=1;n)$$

gdzie krotka $(P.i \mid i=1;n)$ jest rozwiązaniem stałopunktowego układu n równań, w którym i -te równanie ma niżej opisaną postać i odwołuje się i -tej treści procedury (denotacji programu) dpr.i oraz wspólnego środowiska czasu deklaracji śro-cd . Dlaczego to środowisko jest wspólne będzie widoczne w definicji konstruktora denotacji deklaracji.

$$P.i.(\text{pak-w.i}, \text{pak-r.i}).\text{skł-gp} =$$

$$\text{jest-błąd.skł-gp} \rightarrow \text{skł-gp}$$

niech

$$\text{par.i} = (\text{pfo-w.i}, \text{pfo-r.i}, \text{pak-w.i}, \text{pak-r.i})$$

$$(\text{wrt-gp}, \text{'OK'}) = \text{skł-gp}$$

$$\text{wrt-lp} = \text{przekaż-aktualne}(\text{par.i}).(\text{śrt-cd}, \text{wrt-gp}) \quad (\text{wartościowanie lok. pocz.})$$

$$\text{wrt-lp} : \text{Błąd} \rightarrow \text{wrt-lp}$$

niech

$$\text{skł-lp} = (\text{wrt-lp}, \text{'OK'}) \quad (\text{skład lokalny początkowy})$$

$$\text{śro-lp} = (\text{śrt-cd}, \text{śrp-cd}[\text{ide.j/P.j} \mid j=1;n]) \quad (\text{środ. lok. pocz.; cd – czasu dek.})$$

$$\text{sta-lp} = (\text{śro-lp}, \text{wrt-lp}, \text{'OK'}) \quad (\text{stan lokalny początkowy})$$

$$\text{dpr.i.sta-lp} = ? \rightarrow ?$$

niech (wykonanie treści procedury)

$$\text{sta-lk} = \text{dpr.i}(\text{sta-lp}) \quad (\text{stan lokalny końcowy})$$

$$(\text{śro-lk}, (\text{wrt-lk}, \text{błąd})) = \text{sta-lk}$$

$$\text{błąd} \neq \text{'OK'} \rightarrow (\text{wrt-gp}, \text{błąd})$$

niech

$$\text{wrt-gk} = \text{zwróć-referencyjne}(\text{pfo-r.i}, \text{pak-r.i}).(\text{wrt-lk}, \text{wrt-gp})$$

$$\text{wrt-gk} : \text{Błąd} \rightarrow (\text{wrt-gp}, \text{wrt-gk}) \quad (\text{tu wrt-gk jest błędem})$$

$$\text{prawda} \rightarrow (\text{wrt-gk}, \text{'OK'})$$

⁸⁴ Teraz staje się widoczne, dlaczego środowisko nie zostało zaliczone do składowych procedury. Gdyby tak było to konstruktor wieloprocedury mógłby otrzymać z każdym kompletem składowych pojedynczej procedury inne środowisko, gdy tymczasem wszystkie wieloprocedury są definiowane w kontekście wspólnego środowiska ich wspólnej deklaracji.

Tę definicję czytamy tak samo jak definicję konstruktora procedury pojedynczej opisaną w rozdziale 7.3.2. Jedyna istotna różnica to zagnieżdżenie w środowisku procedur czasu deklaracji srp-cd.i wektora procedur $[\text{ide.j/P.j} \mid j=1;n]$ w miejsce jednej procedury.

W chwili wywołania każda z procedur otrzyma to samo środowisko lokalne początkowe $\text{śro-lp} = (\text{śrt-cd}, \text{śrp-cd}[\text{ide.j/P.j} \mid j=1;n])$ oraz swój indywidualny skład globalny początkowy skł-gp . Właśnie dlatego nie uczyniłem środowiska elementem składowej procedury, aby wszystkie składowe wieloprocedury otrzymały to samo środowisko.

Zauważmy też, że dla $n = 1$ nasza definicja jest identyczna jak dla rekurencji pojedynczej, możemy więc przyjąć ją jako uniwersalną definicję konstruktora wieloprocedury dla dowolnego $n \geq 1$.

7.4.3 Instrukcja wywołania składowej wieloprocedury

Wywołanie każdej z procedur wchodzących w skład wieloprocedury nie różni się niczym od wywołania procedury zadeklarowanej jako pojedyncza. Cały mechanizm wieloprocedurowości jest bowiem zaszyty w konstruktorze wieloprocedury. W tym więc przypadku następuje przeniesienie konstrukcji opisanej w rozdziale 7.3.3 bez żadnych zmian.

7.4.4 Deklaracja wieloprocedury

Konstruktor deklaracji wieloprocedury budujemy analogicznie do konstruktora deklaracji pojedynczych (rozdział 7.3.4):

$\text{deklaruj-wiepro-imp} : \text{SkładoweWiePro} \mapsto \text{DenDekWiePri}$

$\text{deklaruj-wiepro-imp} : \text{SkładoweWiePro} \mapsto \text{Stan} \mapsto \text{Stan}$

$\text{deklaruj-wiepro-imp.swp.sta} =$

$\text{jest-błąd.sta} \quad \rightarrow \text{sta}$

niech

$((\text{ide.i}, \text{pfo-r.i}, \text{pfo-w.i}, \text{pro.i}) \mid i = 1;n) = \text{swp}$

$(\text{śro}, \text{wrt}, \text{'OK'}) = \text{sta}$

$(\text{śrt}, \text{śrp}) = \text{śro}$

$\text{są-powtórzenia.}(\text{ide.i} \mid i=1;n) \quad \rightarrow \text{sta} \quad \leftarrow \text{powtórzenia-nazw-procedur}$

$(\exists i).\text{war.}(\text{ide.i}) : \text{zadeklarowane.sta} \quad \rightarrow \text{sta} \quad \leftarrow \text{nazwa-procedury-zajęta}$

niech

$(\text{P.i} \mid i=1;n) = \text{twórz-wiepro-imp.}(\text{swp}, \text{śro})$

$\text{śro-k} = (\text{śrt}, \text{śrp}[\text{ide.i/P.i} \mid i=1;n])$

prawda $\rightarrow (\text{śro-k}, \text{wrt}, \text{'OK'})$

7.5 Procedury funkcyjne

Procedury funkcyjne różnią się od imperatywnych tym, że wynikiem ich wywołania nie jest stan, ale dana utypowiona czyli kompozyt. O ile więc procedury imperatywne można traktować jako instrukcje wzbogacone o parametry, o tyle procedury funkcyjne odpowiadają

wzbogaconym o parametry wyrażeniom danologicznym. W konsekwencji wywołania procedur funkcyjnych będą w **Lingua-2** szczególnym rodzajem takich wyrażeń.

7.5.1 Struktura deklaracji procedury funkcyjnej

Mimo że denotacyjnie procedury funkcyjne odpowiadają wyrażeniom, to treści ich deklaracji będą się składały z programu — być może trywialnego, tj. składającego się z trywialnej preambuły i trywialnej instrukcji — oraz wyrażenia. Program przetwarza stan wejściowy wywołania na stan pośredni, a pośredni przekazuje do wyrażenia, które przetwarza ten stan na kompozyt. Przykładem może być deklaracja procedury, która oblicza moduł zadanej potęgi zadanej liczby:

```
fun moduł-potęgi(n, m real)
  let p be real
  p := 1
  while m > 0 do p := p*n ; m:=m-1 od
  return if p ≤ 0 then -p else p fi as real
end fun
```

W szczególności w deklaracji procedury funkcyjnej po **return** może stać wyrażenie ograniczone do pojedynczego identyfikatora, tj. zmienna, a przed **return** może stać program zredukowany do postaci **skip; skip**. W tym drugim przypadku procedura sprowadza się do wyrażenia.⁸⁵ Wyrażenie stojące w deklaracji po słowie **return** będziemy nazywali *wyrażeniem eksportującym*.

W niektórych językach (np. w Pascalu [48]) dopuszcza się zarówno parametry referencyjne, jak i zmienne globalne procedur funkcyjnych, co oznacza, że wywołanie takiej procedury obok wyliczenia wartości ma jeszcze tzw. *efekt uboczny* (ang. *side effect*) polegający na zmianie stanu. W **Lingua-2** świadomie rezygnuję z tej opcji, gdyż w moim przekonaniu każdy niejawnny efekt działania programu może prowadzić do błędów programisty. Zresztą autorzy Pascala, choć dopuszczali efekty uboczne procedur funkcyjnych, stanowczo odradzali programistom korzystanie z tej możliwości (patrz [48] s. 79). Jest zastanawiające dlaczego wobec tego nie wyeliminowali ich z języka?

W tym miejscu znajduje też wyjaśnienie wykluczenie wyrażeń (innych niż identyfikatory) na miejscu parametrów aktualnych procedury, co zostało wstępnie omówione w rozdziale 7.1.4. Gdybyśmy nie wprowadzili tego wykluczenia, wywołania procedur funkcyjnych mogłyby być parametrami procedur, co prowadziłoby do nowego rodzaju rekurencji i z pewnością komplikowałoby zarówno model denotacyjny, jak i logikę programów (rozdział 8).

Pewnym rozwiązaniem technicznym tego problemu mogłoby być wykluczenie z parametrów aktualnych wywołań procedur funkcyjnych, to jednak oznaczałoby wprowadzenie do języka dwóch kategorii wyrażeń: jednych z wywołaniami procedur funkcyjnych, a drugich — bez. To znów możliwe, ale też prowadzi do komplikacji modelu. Zdecydowałem więc, aby nie dopuszczać nietrywialnych wyrażeń na miejscu parametrów aktualnych.

Kolejne ograniczenie jakie przyjmuję dla deklaracji procedur funkcyjnych, to wykluczenie rekurencji, choć oczywiście będą mogły w nich występować wywołania imperatywnych

⁸⁵ Tę uniwersalną postać deklaracji procedury funkcyjnej zasugerował mi Andrzej Tarlecki.

procedur rekurencyjnych. Opracowanie modelu dla funkcyjnych procedur z rekurencją pozostawiam czytelnikowi jako pożyteczne ćwiczenie.

7.5.2 Dziedziny procedur funkcyjnych

W przypadku procedur funkcyjnych mamy do czynienia z trzema rodzajami obiektów:

- *procedura funkcyjna* — funkcja przekształcająca stan w kompozyt,
- *deklaracja procedury funkcyjnej* — funkcja modyfikująca środowisko,
- *wywołanie procedury funkcyjnej* — funkcja przekształcająca stan w kompozyt.

Zgodnie z uczynioną już zapowiedzią procedury funkcyjne będę niekiedy nazywał „funkcjami”.

Wprowadzenie procedur funkcyjnych do języka wymaga rozbudowania go o dwie nowe dziedziny:

$\text{ddf} : \text{DenDekPrf} = \text{Stan} \mapsto \text{Stan}$ (denotacje deklaracji procedur funkcyjnych)
 $\text{prf} : \text{ProFun} = \text{ParAkt} \mapsto \text{Stan} \rightarrow \text{KompozytB}$ (procedury funkcyjne)

Oczywiście dziedzina denotacji wyrażeń zostaje rozbudowana o wywołania procedur funkcyjnych. Oznacza to dość istotną zmianę polegającą nie tylko na dodaniu nowych denotacji, ale też i na tym, że obecnie denotacje wyrażeń będą sięgały do środowiska w celu odczytania z niego procedur funkcyjnych. Zauważmy, że w **Lingua-1** choć denotacje wyrażeń formalnie otrzymywały stany jako argumenty, to jednak faktycznie „sięgały” jedynie do zawartych w nich wartościowań.

Zauważmy też, że procedury funkcyjne operują na stanach, a nie na składach, jak było w przypadku procedur imperatywnych. Ta zmiana wynika oczywiście z zaniechania rekurencji.

7.5.3 Konstruktor procedury funkcyjnej

Procedurę funkcyjną można potraktować jako sekwencyjne złożenie trzech składowych:

1. funkcji przekazującej parametry aktualne do stanu wywołania procedury,
2. instrukcji,
3. wyrażenia eksportującego.

Dziedzinę *składowych procedur funkcyjnych* definiuję w sposób następujący:

$\text{spf} : \text{SkładoweF} = \text{Identyfikator} \times \text{ParFor} \times \text{DenPro} \times \text{DenWyrDan} \times \text{DenWyrTyp}$

Zatem składowe procedury funkcyjnej obejmują składowe procedury imperatywnej bez parametrów referencyjnych plus denotacje wyrażeń danologicznego i typologicznego:

dwd — denotację wyrażenia eksportującego,

dwt — denotację wyrażenia typologicznego wskazującą na oczekiwany typ eksportowanej wartości.

Rozpocznę od zdefiniowania pomocniczego operatora eksportu:

$\text{eksportuj} : (\text{DenWyrDan} \times \text{DenWyrTyp}) \mapsto \text{Stan} \mapsto \text{KompozytB}$

$\text{eksportuj}(\text{dwd}, \text{dwt}).\text{sta} =$

jest-błąd.sta $\rightarrow$ błąd.sta

niech

typ = dwt.sta (oczekiwany typ kompozytu)

kom = dwd.sta (eksportowany kompozyt)

typ : Błąd $\rightarrow$ typ

kom : Błąd $\rightarrow$ kom

niech

(kor-t, jar) = typ

(dan, kor-k) = kom

nie (kor-t koherentny kor-k) $\rightarrow$ 'niezgodność-korpusów'

jar.kom $\neq$ (tt, ('boolean')) $\rightarrow$ 'niespełnione-jarzmo'

prawda $\rightarrow$ kom

Ten operator zwraca kompozyt wyliczany przez denotację wyrażenia eksportującego dwd, pod warunkiem że korpus tego kompozytu jest koherentny z korpusem wyliczonego typu, a kompozyt spełnia jarzmo tego typu.

Teraz możemy zdefiniować konstruktor procedur funkcyjnych odwołujący się do funkcji przekazywania parametrów aktualnych i do operatora eksportu

twórz-pro-fun : SkładoweF $\mapsto$ ProFun

lub w postaci rozwiniętej:

twórz-pro-fun : Identyfikator x ParFor x DenPro x DenWyrDan x DenWyrTyp
 $\mapsto$ (ParAkt $\mapsto$ Stan $\rightarrow$ KompozytB)

Zauważmy, że ten konstruktor nie przyjmuje środowiska jako drugiego argumentu (jak to miało miejsce w przypadku procedur imperatywnych), gdyż procedury funkcyjne działają na stanach, a nie na składach. To oczywiście konsekwencja zrezygnowania z rekurencji. Wartością naszego konstruktora jest funkcja:

F : ParAkt $\mapsto$ Stan $\rightarrow$ KompozytB

F = twórz-pro-fun.(ide-n, pfo-w, dpr, dwd, dwt)

zdefiniowana w następujący sposób:

F.pak.sta-gp = (gp — globalny początkowy)

jest-błąd.sta $\rightarrow$ błąd.sta

niech

((śrt, śrp), (wrt, 'OK')) = sta-gp

wrt-lp = przekaz-aktualne.(pfo-w, (), pak-w, ()).(śrt, wrt) (lp — lokalny pocz.)

wrt-lp : Błąd $\rightarrow$ wrt-lp

niech

sta-lp = ((śrt, śrp), (wrt-lp, 'OK'))

sta-lk = dpr.sta-lp

jest-błąd.sta-lk $\rightarrow$ błąd.sta-lk

prawda $\rightarrow$ eksportuj.(dwd, dwt).sta-lk

Przy pomocy operatora przekazywania parametrów aktualnych tworzymy stan lokalny początkowy, do którego jest aplikowany program będący składową treści procedury. W stanie wynikowym tego programu jest wykonywane wyrażenie eksportujące i tak powstały kompozyt jest kompozytem końcowym wykonania wywołania procedury funkcyjnej. Korpus tego kompozytu musi być typu wskazanego przez wyrażenie typologiczne, o co dba operator eksportowania. Puste krotki () występujące wśród argumentów funkcji przekazywania parametrów odpowiadają parametrom referencyjnym odpowiednio formalnym i aktualnym.

Zauważmy, że równanie definiujące funkcję F nie jest równaniem stałopunktowym, jak to miało miejsce w przypadku procedur imperatywnych. To właśnie konsekwencja zrezygnowania z rekurencji. Praktycznie oznacza to, że jeżeli w deklaracji procedury funkcyjnej pojawi się jej wywołanie — czego oczywiście składniowo nie możemy wykluczyć — to doprowadzi ono do wygenerowania komunikatu błędu ‘procedura-niezadeklarowana’.

7.5.4 Wyrażenie wywołania procedury funkcyjnej

Procedura funkcyjna to funkcja, która po otrzymaniu aktualnych parametrów wartościowych oddaje denotację wyrażenia danologicznego. Na wywołanie procedury funkcyjnej składają się więc cztery kroki:

1. pobranie procedury ze środowiska,
2. wyliczenie wartości jej parametrów, tj. wartości identyfikatorów będących aktualnymi parametrami wywołania,
3. zastosowanie procedury do parametrów wartościowych w wyniku czego powstaje denotacja wyrażenia,
4. zastosowanie tak powstałej denotacji wyrażenia do aktualnego stanu, w wyniku czego — jeżeli wykonanie wyrażenia się zakończy — powstaje kompozyt lub komunikat błędu.

Tak więc konstruktor wywołania jest typu

$\text{wołaj-pro-fun} : \text{Identyfikator} \times \text{ParAkt} \mapsto \text{DenWyrDan}$

czyli:

$\text{wołaj-pro-fun} : \text{Identyfikator} \times \text{ParAkt} \mapsto \text{Stan} \rightarrow \text{KompozytB}$

lub jeszcze inaczej:

$\text{wołaj-pro-fun}.\text{(ide, pak)} : \text{Stan} \rightarrow \text{KompozytB}$

Denotację wyrażenia powstającego w wyniku wywołania procedury definiuję w sposób następujący:

$\text{wołaj-pro-fun}.\text{(ide, pak).sta} =$

jest-błąd.sta $\rightarrow$ błąd.sta

niech

$\text{((śrt, śrp), skł)} = \text{sta}$

$\text{śrp.ide} = ? \rightarrow \text{'procedura-niezadeklarowana'}$

$\acute{s}rp.ide : \text{Prolmp} \rightarrow \text{'procedura-niefukcyjna'}$

niech

$\text{prf} = \acute{s}rp.ide$ (procedura funkcyjna)

$\text{prf.pak.sta} = ? \rightarrow ?$

prawda $\rightarrow \text{prf.pak.sta}$

Jeśli stan nie niesie komunikat błędu, a w środowisku została zadeklarowana procedura funkcyjna o nazwie *ide*, to procedura jest aplikowana do aktualnych parametrów wartościowych i aktualnego stanu i o ile to obliczenie się zakończy, to otrzymujemy kompozyt końcowy wywołania lub błąd.

7.5.5 Deklaracja procedury funkcyjnej

W tym przypadku konstruktor jest funkcją typu:

$\text{deklaruj-pro-fun} : \text{SkładowePrf} \mapsto \text{Stan} \mapsto \text{Stan}$

czyli

$\text{deklaruj-pro-fun} : \text{Identyfikator} \times \text{ParFor} \times \text{DenPro} \times \text{DenWyrDan} \times \text{DenWyrTyp} \mapsto \text{Stan} \mapsto \text{Stan}$

$\text{deklaruj-pro-fun.spf.sta} =$

$\text{jest-błąd.sta} \rightarrow \text{błąd.sta}$

niech

$(ide, pfo, dpr, dwd, dwt) = \text{spf}$

$((\acute{s}rt, \acute{s}rp), \text{skł}) = \text{sta}$

$ide : \text{zadeklarowane.sta} \rightarrow \text{sta} \leftarrow \text{'identyfikator-ide-zajęty'}$

niech

$F = \text{twórz-pro-fun.spf}$

prawda $\rightarrow ((\acute{s}rt, \acute{s}rp[ide/F]), \text{skł})$

7.6 Procedury jako parametry procedur

Jak już wiemy, próba zdefiniowania takiego mechanizmu procedur, w którym procedury mogą być parametrami innych procedur prowadzi w przypadku ogólnym do niedozwolonej rekurencji typu:

$\text{Procedura} = \text{Parametr} \mapsto \text{Skład} \rightarrow \text{Skład}$

$\text{Parametr} = \text{Kompozyt} \mid \text{Procedura}$

Tego typu mechanizm był zrealizowany w Algolu 60, to jednak wymagało posłużenia się modelami niedenotacyjnymi, a przynajmniej niedenotacyjnymi na gruncie teorii mnogości. Na poziomie modelu oznaczało to natomiast, że procedura, która miała wziąć inną procedurę jako parametr, lub nawet samą siebie, w rzeczywistości „otrzymywała” tekst tej procedury, a nie ją samą.

Fakt, że powyższy układ równań nie ma rozwiązania, nie oznacza jednak, że funkcje nie mogą przyjmować innych funkcji jako swoich argumentów. Jest to możliwe, należy jednak zbudować odpowiednią hierarchię funkcji, aby żadna z nich nie mogła wziąć jako argumentu samej siebie, czy to bezpośrednio, czy też pośrednio. Taka konstrukcja została opisana w [29] i w uproszczonej wersji ma następującą postać:

$$\text{Procedura.0} = \text{Parametr.0} \mapsto \text{Stan} \rightarrow \text{Stan}$$

$$\text{Parametr.0} = \text{Kompozyt}$$

Dla $n > 0$:

$$\text{Procedura.n} = \text{Parametr.n} \mapsto \text{Stan} \rightarrow \text{Stan}$$

$$\text{Parametr.n} = \text{Parametr.0} \mid \dots \mid \text{Parametr.(n-1)}$$

Zatem procedura danego poziomu może brać jako argumenty proceduralne jedynie procedury niższych poziomów. Tego wątku nie będę jednak dalej rozwijał i nie wprowadzę go do języka **Lingua** by nie komplikować modelu.

7.7 Programy

W **Lingua-1** programy składały się z dwóch następujących po sobie członów: preambuły i instrukcji. W **Lingua-2** będzie podobnie, z tym że preambuła będzie teraz (dowolnie skomponowaną) sekwencją deklaracji procedur, deklaracji zmiennych oraz definicji typów. To założenie — jak już nie raz przy budowaniu naszego języka — ma charakter pragmatyczny, a jego celem jest prostota reguł konstruowania programów poprawnych (rozdz.8). Wymienione w rozdziale 6.1.9 konstruktory preambuł uzupełniam o trzy kolejne odpowiadające deklaracjom procedur i funkcji

$$\text{twórz-pream-z-dek-pri} : \text{DenDekPri} \mapsto \text{DenPre}$$

$$\text{twórz-pream-z-dek-wie-pri} : \text{DenDekWiePri} \mapsto \text{DenPre}$$

$$\text{twórz-pream-z-dek-prf} : \text{DenDekPrf} \mapsto \text{DenPre}$$

Wszystkie one tworzą preambułę z odpowiednich deklaracji.

7.8 Składnia i semantyka

7.8.1 Sygnatura algebry denotacji

Jak pamiętamy z rozdziałów 4.2 i 6.2.2 składnię konkretną języka budujemy z jego składni abstrakcyjnej, którą z kolei można algorytmicznie wygenerować z sygnatury algebry denotacji. Zacznę zatem od napisania tej sygnatury w sposób wyraźny. Jest ona w dużej mierze wyznaczona przez wcześniej podane definicje konstruktorów, jednakże:

- nie wszystkie z nich są konstruktorami w algebrze denotacji — niektóre służą jedynie do zdefiniowania tych ostatnich,
- nie ma wśród nich niektórych konstruktorów potrzebnych do zbudowania części osiągalnej przyszłej algebry.

Analogiczna sytuacja dotyczy też dziedzin denotacyjnych.

7.8.1.1 Nośniki algebry denotacji

Poniższa lista dziedzin obejmuje wszystkie dziedziny algebry denotacji języka **Lingua-2**, a więc również i te, które występują w **Lingua-1**:

ide	: Identyfikator	(identyfikatory)
dwd	: DenWyrDan	(denotacje wyrażeń danologicznych, w tym wywoł. proc. fun.)
dwt	: DenWyrTyp	(denotacje wyrażeń typologicznych)
din	: DenIns	(denotacje instrukcji, w tym wywołania proc. imperatywnych)
pfo	: ParFor	(parametry formalne)
pak	: ParAkt	(parametry aktualne)
spi	: SkładowePri	(składowe procedury imperatywnej)
swp	: SkładoweWiePro	(składowe wieloprocedury imperatywnej)
sff	: SkładowePrf	(składowe procedury funkcyjnej)
ddz	: DenDekZmi	(denotacje deklaracji zmiennej)
ddt	: DenDefTyp	(denotacje definicji typów)
ddp	: DenDekPri	(denotacje deklaracji procedur imperatywnych)
ddw	: DenDekWiePri	(denotacja deklaracji wieloprocedury)
ddf	: DenDekPrf	(denotacje deklaracji procedur funkcyjnych)
dpe	: DenPre	(denotacje preambuł)
dpr	: DenPro	(denotacje programów)

7.8.1.2 Konstruktory algebry denotacji

Poniższa lista obejmuje jedynie te konstruktory w **Lingua-2**, które nie mają swoich odpowiedników w **Lingua-1**. Nie pokrywa się ona jednak z listą konstruktorów zdefiniowanych w niniejszym rozdziale 7 gdyż:

1. nie zawiera konstruktorów pomocniczych takich jak np. konstruktory tworzenia procedur, które służą jedynie do zdefiniowania konstruktorów deklaracji i wywołań, ale nie należą do algebry denotacji,
2. zawiera pewne nowe konstruktory służące do generowania części osiągalnej nośników algebry denotacji.

Wyrażenia będące wywołaniami procedur funkcyjnych

wołaj-pro-fun : Identyfikator $\times$ ParAkt $\mapsto$ DenWyrDan

Definicja w rozdz.7.5.4.

Instrukcje będące wywołaniami procedur imperatywnych

wołaj-pro-imp : Identyfikator x ParAkt x ParAkt $\mapsto$ DenIns

Definicja w rozdz.7.3.3.

Parametry aktualne

twórz-pusty-akt : $\mapsto$ ParAkt

twórz-par-akt : Identyfikator $\mapsto$ ParAkt

sekwencja-par-akt : ParAkt x ParAkt $\mapsto$ ParAkt

Definicje w rozdziale 7.3.1

Parametry formalne

twórz-pusty-for : $\mapsto$ ParFor

twórz-par-for : Identyfikator x DenWyrTyp $\mapsto$ ParFor

sekwencja-par-for : ParFor x ParFor $\mapsto$ ParFor

Definicje w rozdziale 7.3.1

Składowe procedury imperatywnej

skł-pro-imp : Identyfikator x ParFor x ParFor x DenPro $\mapsto$ SkładowePri

skł-pro-imp.(ide, pfo-r, pfo-w, dpr) = (ide, pfo-r, pfo-w, dma)

Ten konstruktor wygląda jak funkcja identycznościowa, jednakże nią nie jest. Komentarz na temat podobnej funkcji został umieszczony na końcu rozdziału 7.3.1.

Składowe wieloprocedury imperatywnej

skł-wie-pro-imp : SkładowePri $\mapsto$ SkładoweWiePro

sekwencja-swi : SkładoweWiePro x SkładoweWiePro $\mapsto$ SkładoweWiePro

Te funkcje pozwalają na tworzenie niepustych list (krotek) składowych procedur imperatywnych. Elementami tych list (krotek) są więc krotki niższego poziomu.

skł-wie-pro-imp.spi = (spi) (krotka jednoelementowa)

sekwencja-swi.(swp1,swp2) = swp1 © swp2 (konkatenacja krotek)

W pierwszym z tych równań mamy do czynienia z jednoelementową krotką (spi), której elementem jest czteroelementowa krotka spi.

Składowe procedury funkcyjnej

skł-pro-fun : Identyfikator x ParFor x DenPro x DenWyrDan x DenWyrTyp $\mapsto$ SkładoweF

skł-pro-fun-if.(ide-n, pfo, dpr, dwd, dwt) = (ide-n, pfo, dpr, dwd, dwt)

Deklaracje procedur imperatywnych

deklaruj-pro-imp : Identyfikator x ParFor x ParFor x DenPro $\mapsto$ DenDekPri

Definicja w rozdz.7.3.4.

Deklaracje wieloprocedur imperatywnych

deklaruj-wiepro-imp : SkładoweWiePro $\mapsto$ DenDekWiePri

Definicja w rozdz.7.4.4

Deklaracje procedur funkcyjnych

deklaruj-pro-fun : Identyfikator x ParFor x DenPro x DenWyrDan x DenWyrTyp $\mapsto$
DenDekPrf

Preambuły

twórz-preambułę-z-dek-pri	: DenDekPri	$\mapsto$ DenPre
twórz-preambułę-z-dek-wie-pri	: DenDekWiePri	$\mapsto$ DenPre
twórz-preambułę-z-dek-prf	: DenDekPrf	$\mapsto$ DenPre
twórz-preambułę-z-def-typu	: DenDefTyp	$\mapsto$ DenPre
twórz-preambułę-z-dek-zmi	: DenDekZmi	$\mapsto$ DenPre
sekwencja-pab	: DenPre x DenPre	$\mapsto$ DenPre

Programy

twórz-program-z-instrukcji	: DenIns	$\mapsto$ DenPro
twórz-prog	: DenPre x DenIns	$\mapsto$ DenPro

7.8.2 Składnia konkretna

W procesie tworzenia składni naszego języka pomijam etap budowania składni abstrakcyjnej, gdyż ma on charakter algorytmiczny, został też szczegółowo opisany dla **Lingua-1** w rozdziale 6.2. Nasze postępowanie w tym przypadku można porównać do postępowania matematyka, który pisząc dowód twierdzenia nie konstruuje go w sposób sformalizowany jako ciąg formuł wyprowadzanych jedno z drugich za pomocą reguł dedukcji, ale posługuje się matematyką intuicyjną, którą jednak można zawsze dostatecznie głęboko sformalizować, aby upewnić się, że prowadzone rozumowanie jest poprawne. Podobnie jak dowody intuicyjne mają swoją bazę teoretyczną w dowodach sformalizowanych, tak tworzenie składni konkretnej i semantyki bezpośrednio z algebry denotacji ma swoją bazę teoretyczną w konstrukcjach opisanych w rozdziałach 2.12 i 4.3

W odróżnieniu od rozdziału 7.8.1.2 gdzie wymieniłem jedynie nowe konstruktory, tym razem podaję pełen opis składni, choć bez powtarzania tych klauzul, które są zaczerpnięte z **Lingua-A** i **Lingua-1**.

ide	: Identyfikator	= (jak w Lingua-A)
wyt	: WyrTyp	= (jak w Lingua-A)
wyd	: WyrDan =	
	(jak w Lingua-A)	
	Identyfikator (ParAktualne)	(wywołanie procedury funkcyjnej)

Zwracam uwagę czytelnika, że **ParAktualne** oznacza język będący nośnikiem algebry składni konkretnej, podczas gdy **ParAkt** jest nośnikiem algebry denotacji.

```
paks : ParAktualne =
    empty-ap |
    Identyfikator |
    ParAktualne , ParAktualne
```

W tej klauzuli **empty-ap** jest słowem kluczowym, którego denotacją jest pusta lista parametrów aktualnych. Zauważmy, że nasza gramatyka dopuszcza takie „dziwaczne” listy parametrów jak np.

```
empty-ap , empty-ap    lub
x, y, z, empty-ap
```

Jest to cena, którą płacimy za prostotę naszej gramatyki. Gdybyśmy jej chcieli uniknąć, to musielibyśmy posłużyć się gramatyką o dwóch równaniach:

```
ParAktualne =
    empty-ap          |
    ParAktualneNiepuste

ParAktualneNiepuste =
    Identyfikator          |
    ParAktualneNiepuste , ParAktualneNiepuste
```

Taka gramatyka prowadzi jednak do algebry niepodobnej do algebry denotacji. Oczywiście tę ostatnią moglibyśmy zmienić, oznaczałoby to jednak, że budując algebrę denotacji musimy myśleć o przyszłej składni, a tego właśnie chcemy uniknąć. Godzimy się więc na opisany wyżej kompromis. Zauważmy jednak, że nasza gramatyka pozwala na generowanie list, których byśmy oczekiwali

```
x, y, z
```

a także, że listy nazwane dziwaczными mają „rozsądne” denotacje.

Te same uwagi odnoszą się do kolejnej klauzuli gramatycznej:

```
pfos : ParFormalne =
    empty-fp
    Identyfikator as WyrTyp |
    ParFormalne , ParFormalne
```

Przykładem listy parametrów formalnych może być:

`x as number, y as boolean, z as pracownik`
gdzie `pracownik` jest typem zdefiniowanym przez użytkownika.

`dez : DekZmi = (jak w Lingua-1)`

`det : DefTyp = (jak w Lingua-1)`

`ins : Instrukcja =`

`(jak w Lingua-1) |`

`call Identyfikator (ref ParAktualne val ParAktualne)`

`spi : Składowa-pi =`

`proc Identyfikator (val ParFormalne ref ParFormalne) Program endproc`

`dpi : DekProImp =`

`Składowa-pi`

Na podstawie twierdzeń z rozdziału 2.3 o redukcji równań stałopunktowych możemy w ostatnim równaniu zastąpić `Składowa-pi` przez prawą stronę definiującego ją równania otrzymując wyraźniejszą definicję deklaracji procedury imperatywnej:

`dpi : DekProImp =`

`proc Identyfikator (val ParFormalne ref ParFormalne)`

`Program`

`end proc`

Składowe wieloprocedur są listami składowych procedur imperatywnych, stąd więc:

`swp : Składowa-wpi =`

`Składowa-pic+`

a ponieważ składowe są składniowo tym samym, co deklaracje, możemy napisać

`dwi : DekWieProImp =`

`begin multiproc`

`DekProImpc+`

```
end multiproc
```

Stąd podobnie jak poprzednio możemy wygenerować ostateczną definicję składni deklaracji wieloprocedury:

```
dwi : DekWieProImp =
  begin multiproc
    [ proc Identyfikator (val ParFormalne ref ParFormalne)
      Program
    endproc ]c+
  end multiproc
```

Oczywiście nawiasy kwadratowe należą do metapoziomu. W analogiczny sposób dochodzimy do klauzuli dla deklaracji procedur funkcyjnych:

```
dpf : DekProFun =
  fun Identyfikator (ParFormalne)
    Program
    return Identyfikator as WyrTyp
  end fun
```

```
pab: Preambuła =
  DeklaracjaProImp      |
  DeklaracjaWieProImp   |
  DeklaracjaProFun      |
  DefinicjaTyp          |
  Deklaracja Zmi        |
  skip                 |
  Preambuła ; Preambuła
```

W tej klauzuli zostały opuszczone nazwy konstruktorów oraz nawiasy związane ze średnikiem. Pierwsza transformacja ma charakter izomorficzny, gdyż pierwsze pięć kategorii występujących w tej klauzuli odpowiadają rozłącznym językom, a to dzięki ujmowaniu ich w nawiasy typu **proc...end proc** i podobne. Druga transformacja jest oczywiście sklejająca, jest jednak dopuszczalna ze względu na łączność operacji składania funkcji. Więcej na temat dopuszczalności takiego homomorfizmu w rozdziale 6.2.2.

```

prg : Program =
  begin-program Instrukcja end-program |
    begin-program Preambuła ; Instrukcja end-program

```

7.8.3 Składnia kolokwialna

W **Lingua-2** dopuszczam wszystkie kolokwializmy **Lingua-1** i dodaję jeden dotyczący parametrów formalnych w nagłówkach deklaracji procedur obu typów. Dopuszczam mianowicie grupowania parametrów w listy o jednakowym typie jak w poniższym przykładzie:

```

proc nazwa (val w, z as real ref x, y as real a, b, c as pracownik)

```

7.8.4 Semantyka

Ponieważ **Lingua-2** semantycznie pokrywa się z **Lingua-1** tam, gdzie pokrywają się ich składnie, w niniejszym rozdziale pokażę semantykę jedynie tych konstrukcji, których brak w **Lingua-1**. W dalszym ciągu będę pisał *ide* zamiast *Sid.[ide]* ze względu na identycznościowy charakter semantyki identyfikatorów. Posłużę się też algebraicznym zapisem semantyki (patrz rozdział 5.6).

7.8.4.1 Parametry aktualne

$\text{Spak} : \text{ParAktualne} \mapsto \text{ParAkt}$ czyli
 $\text{Spak} : \text{ParAktualne} \mapsto \text{Identyfikator}^{c*}$

$\text{Spak}[\text{ide}] = (\text{ide})$
 $\text{Spak}[\text{pak-1} \text{ , pak-2}] = \text{Spak}[\text{pak-1}] \odot \text{Spak}[\text{pak-2}]$

7.8.4.2 Parametry formalne

$\text{Spfo} : \text{ParFormalne} \mapsto \text{ParFor}$ czyli
 $\text{Spfo} : \text{ParFormalne} \mapsto (\text{Identyfikator} \times \text{DenWyrTyp})^{c*}$
 $\text{Spfo}[\text{ide as wyt}] = ((\text{ide}, \text{Swty}[\text{wyt}]))$
 $\text{Spfo}[\text{paks-1} \text{ , paks-2}] = \text{Spfo}[\text{paks-1}] \odot \text{Spfo}[\text{paks-2}]$

7.8.4.3 Wyrażenia danologiczne: wywołanie procedury funkcyjnej

$\text{Swd} : \text{WyrDan} \mapsto \text{DenWyrDan}$
 $\text{Swd}[\text{ide (paks)}] = \text{wołaj-pro-fun}(\text{ide}, \text{Spak}[\text{paks}])$

7.8.4.4 Instrukcje: wywołanie procedury imperatywnej

$\text{Sin} : \text{Instrukcje} \mapsto \text{DenIns}$

$\text{Sin}[\text{call } \text{ide } (\text{ref } \text{pak-r } \text{val } \text{pak-w})] =$
 $\text{wołaj-pro-imp}(\text{ide}, \text{Spak}[\text{pak-r}], \text{Spak}[\text{pak-w}])$

7.8.4.5 Deklaracja procedury imperatywnej

$\text{Sdpi} : \text{DekProlmp} \mapsto \text{DenDekPri}$
 $\text{Sdpi}[\text{proc } \text{ide } (\text{val } \text{pfo-w } \text{ref } \text{pfo-r}) \text{ pro } \text{end proc}] =$
 $\text{deklaruj-pro-imp}(\text{ide}, \text{Spfo}[\text{pfo-r}], \text{Spfo}[\text{pfo-w}], \text{Spr}[\text{pro}])$

7.8.4.6 Deklaracja wieloprocedury imperatywnej

$\text{Sdwpi} : \text{DekWieProlmp} \mapsto \text{DenDekWiePri}$
 $\text{Sdwpi}[\text{begin multiproc}$
 $\quad (\text{proc } \text{ide.i } (\text{val } \text{pfo-w.i } \text{ref } \text{pfo-r.i}) \text{ pro.i } \text{end proc } | i=1;n)$
 $\text{end multiproc}] =$
 $\text{deklaruj-wiepro-imp}((\text{ide-i}, \text{Spfo}[\text{pfo-w.i}], \text{Spfo}[\text{pfo-r.i}], \text{Spr}[\text{pro.i}]) | i=1;n)$

7.8.4.7 Deklaracje procedury funkcyjnej

$\text{Sdpf} : \text{DekPrf} \mapsto \text{DenDekPrf}$
 $\text{Sdpf}[\text{fun } \text{ide-n } (\text{pfo}) \text{pro } \text{return } \text{wyr-r } \text{as } \text{wyt } \text{end fun}] =$
 $\text{if-deklaruj-pro-fun}(\text{ide-n}, \text{Spfo}[\text{pfo}], \text{Spr}[\text{pro}], \text{Dwd}[\text{wyr-r}], \text{Swty}[\text{wyt}])$

7.8.4.8 Preambuły

$\text{Spre} : \text{Preambuła} \mapsto \text{DenPre}$
 $\text{Spre}[\text{dez}] = \text{Sdz}[\text{dez}] \quad (\text{deklaracja zmiennych})$
 $\text{Spre}[\text{det}] = \text{Sdt}[\text{det}] \quad (\text{definicja typów})$
 $\text{Spre}[\text{dpi}] = \text{Sdpi}[\text{dpi}] \quad (\text{deklaracja procedury imperatywnej})$
 $\text{Spre}[\text{dwpi}] = \text{Sdwpi}[\text{dwpi}] \quad (\text{deklaracja wieloprocedury imperatywnej})$
 $\text{Spre}[\text{dpf}] = \text{Sdpf}[\text{dpf}] \quad (\text{deklaracja procedury funkcyjnej})$
 $\text{Spre}[\text{pab} ; \text{pab}] = \text{Spre}[\text{pab}] \bullet \text{Spre}[\text{pab}]$

7.8.4.9 Programy

$\text{Spr} : \text{Program} \mapsto \text{DenPro}$ czyli

$\text{Spr.}[\text{ins}] = \text{Sin.}[\text{ins}]$

$\text{Spr.}[\text{pab} ; \text{ins}] = \text{Spre.}[\text{pab}] \bullet \text{Sin.}[\text{ins}]$

8 Lingua-2V — programowanie walidujące

Przez programowanie walidujące rozumiem taką metodę budowania programów, która wraz z programem tworzy jego matematycznie adekwatną specyfikację. O tej metodzie pisałem ogólnie w rozdziale 1.1, a w rozdziale 3 zarysowałem jej fundament matematyczny oparty na pojęciach częściowej i całkowitej poprawności programu. W niniejszym rozdziale pokażę zasady wyposażania języka rodziny **Lingua** w narzędzia służące do programowania walidującego oraz zilustruję te zasady na przykładzie języka **Lingua-2**.

Ideę programowania walidującego (bez procedur) naszkicowałem w pracach [17], [18] i [19] opublikowanych na przełomie lat 1970/1980. Na gruncie przedstawionych tam badań doszedłem do wniosku, że aby moje idee sprowadzić do poziomu praktycznego, trzeba zbudować język programowania z matematycznie zdefiniowaną składnią i semantyką, gdyż jedynie na takim gruncie można myśleć o udowodnieniu poprawności reguł budowania programów. Matematycznym modelom języków programowania poświęciłem więc następne dziesięć lat, aby w roku 1990 — o czym już pisałem — zająć się budowaniem mojej rodzinnej firmy, co zajęło mi kolejne lat dwadzieścia jeden. Dopiero więc w roku 2013 na nowo podjąłem prace nad moim projektem, do którego wstępem ma się stać niniejsza książka.

8.1 Struktura języka

Najkrócej rzecz ujmując język programowania walidującego to język tez, które będziemy nazywać *metaprogramami*. Każdy metaprogram składa się z dwóch przeplatających się wzajemnie warstw:

1. *warstwy programistycznej*, która jest programem w zwykłym sensie,
2. *warstwy deskryptywnej*, na którą składają się pre-warunek, post-warunek oraz asercje (warunki) zagnieżdżone pomiędzy instrukcjami programu.

Metaprogram nazywamy *poprawnym*, jeżeli zawarty w nim program jest całkowicie poprawny (rozdział 3.4) względem pre- i post-warunku, a asercje są spełnione w trakcie jego wykonywania. W procesie budowania programu asercje służą do podejmowania decyzji, które konstruktory programów mogą być na danym etapie zastosowane.

Idea programowania walidującego polega na tym, aby z wyjściowych prostych programów poprawnych, których poprawność dowodzimy w sposób tradycyjny, tworzyć złożone programy poprawne za pomocą reguł gwarantujących poprawność. Tak więc tradycyjne dowodzenie poprawności ma miejsce jedynie na początku powstawania programu, natomiast na dalszych etapach poprawności jest gwarantowana przez użycie odpowiednich reguł. Mamy tu sytuację analogiczną do teorii sformalizowanej, na którą można spojrzeć jako na język tez wyrażających własności określonych obiektów matematycznych, który został wyposażony w reguły dedukcji pozwalające na wywodzenie z aksjomatów teorii lub z twierdzeń już udowodnionych, ich logicznych konsekwencji.

W niniejszym rozdziale pokażę — na przykładzie **Lingua-2** — jak dla zadanego *źródłowego języka* imperatywnego **Lingua-n** zbudować odpowiadający mu język programowania

walidującego **Lingua-nV**⁸⁶, który zawiera w sobie cały język źródłowy plus trzy dodatkowe kategorie składniowe warstwy deskryptywnej:

1. *Warunki*, których denotacjami są wielowartościowe predykaty częściowe na stanach.
2. *Instrukcje specyfikowane*, których denotacjami są funkcje częściowe na stanach (jak denotacje instrukcji), i w których warstwa deskryptywna opisuje własności warstwy programistycznej.
3. *Tezy*, których denotacjami są klasyczne wartości boolowskie pp lub ff i które dzielą się na trzy podkategorie:
 - 3.1. *właściwości* — wyrażające składniowe własności programów, np. mówiące, że dana deklaracja procedury występuje w preambule danego metaprogramu,
 - 3.2. *metawarunki* — wyrażające semantyczne własności warunków, np. mówiące, że dany warunek nigdy nie jest fałszywy, ale może być nieokreślony,
 - 3.3. *metaprogramy* — wyrażające własności całkowitej poprawności zawartych w nich programów.

Do właściwości i metawarunków będziemy stosowali klasyczne spójniki zdaniotwórcze oraz klasyczne identyfikatory, gdyż na poziomie opisu własności programów pozostajemy w obszarze logiki klasycznej. Predykaty nieklasyczne pojawiają się jedynie na poziomie denotacji programów, a więc na poziomie ich wykonywania.

Wbrew przyjętej w książce filozofii *od denotacji do składni*, przy budowaniu języka programowania walidacyjnego postępuję w kolejności *od składni do denotacji*. Czynn timerak dlatego, że tym razem punktem wyjścia jest składnia konkretna języka źródłowego, która powinna być w całości zanurzona w języku walidacyjnym.

8.2 Warunki

By nie wchodzić w uciążliwe technikalnia, klasy *warunków* nie będę definiował w sposób szczegółowy, a jedynie założę, że ma określone własności. Z opisu tych własności powinno już wynikać, jak dla każdego konkretnego języka źródłowego z rodziny **Lingua** zbudować odpowiadającą mu kategorię warunków.

Poszczególne typy warunków będę opisywał odwołując się do ich (antycypowanej) składni konkretnej. Mam nadzieję, że to przyczyni się do czytelności wykładu nie naruszając zbytnio rygorów jego matematycznej precyzji.

Przy definiowaniu semantyki warunków posłużę się następującymi skrótami definicyjnymi dotyczącymi kompozytów boolowskich:

kp = (pp, ('boolowska'))

kf = (ff, ('boolowska'))

i założę, że spójniki logiczne McCarthy'ego są zdefiniowane na takich kompozytach zgodnie z filozofią rachunku tegoż autora.

⁸⁶ Przyjmuję przyrostek „V” od „validating”, a nie „W” od walidujące, gdyż składnię **Lingua** buduję na bazie języka angielskiego.

8.2.1 O warunkach ogólnie

Zakładam, że w każdym języku **Lingua-nV** zostanie zdefiniowana składniowa dziedzina warunków:

```

wrn : Warunek =
    WarDan   |                               (warunki danologiczne)
    WarWal   |                               (warunki walidacyjne)
    Instrukcja @ Warunek |                   (warunki algorytmiczne)
    (Warunek and Warunek) | (Warunek or Warunek) | (not Warunek) |
    ( $\forall$  Identyfikator, Warunek) | ( $\exists$  Identyfikator, Warunek)

```

Intuicyjnie i praktycznie *warunki danologiczne* odpowiadają wyrażeniom boolowskim zbudowanym nad rozszerzonym repertuarem konstruktorów danych, a więc zawierającym konstruktory, które pozwalają na wyrażanie własności danych niekoniecznie wyrażalnych w języku źródłowym. Np. na poziomie warunków danologicznych możemy mieć dostępny konstruktor uporządkowana-lista, który może nie być dostępny na poziomie programów. Warunki danologiczne będą więc stanowić pewien nadzbiór klasy wyrażeń boolowskich. W konsekwencji, co wynika z przedstawionej wyżej klauzuli gramatycznej, taki nadzbiór będzie też tworzyła kategoria Warunek. Jednakże — jak było to wyjaśnione na końcu rozdziału 5.3.5 — wyodrębnienie wyrażeń boolowskich (a więc i warunków) jako niezależnego nośnika algebry składni wiąże się z rozwiązaniami niewygodnymi z punktu widzenia programisty. Przyjmiemy zatem, że warunki stanowią nadzbiór kategorii składniowej wszystkich wyrażeń danologicznych, a nie jedynie wyrażeń boolowskich, oraz, że ich semantyka jest funkcją:

$Swa : Warunek \mapsto Stan \rightarrow Kompozyt \mid Błąd$

Zauważmy, że denotacjami warunków są funkcje częściowe, co jest związane z założeniem, że ich klasa obejmuje wszystkie wyrażenia danologiczne. O boolowskich konstruktorach warunków zakładam, że są zdefiniowane zgodnie z filozofią McCarthy’ego, czyli analogicznie jak dla wyrażeń danologicznych (rozdział 5.3.2). Ich definicji nie będę więc tu powtarzał. Z kolei kwantyfikatory są zdefiniowane zgodnie z rachunkiem Kleene’ego, o którym również była mowa w rozdziale 2.9. Ich semantykę definiują dwa poniższe równania:

$\forall : Identyfikator \times Warunek \mapsto Warunek$

$Swa.[\forall (ide, wrn)].sta =$

jest-błąd.sta $\rightarrow$ błąd.sta

niech

$(śro, (wrt, 'OK')) = sta$

dla każdej $war : Wartość$, $Swa.[wrn].(śro, (wrt[ide/war], 'OK')) = kp \rightarrow kp$

istnieje $war : Wartość$, $Swa.[wrn].(śro, (wrt[ide/war], 'OK')) = kf \rightarrow kf$

prawda $\rightarrow$ 'nigdy-fałsz'

Komunikat 'nigdy-fałsz' odpowiada sytuacji, gdy kompozyt

$Swa.[wrn].(śro, (wrt[ide/war], 'OK'))$

nigdy nie jest *kf*, ale też nie zawsze jest *kp*, tzn. gdy jest:

- albo *kp*,
- albo błędem,
- albo nieokreślona,

przy czym dla co najmniej jednej wartości *war* nie jest równe *kp*. Definicja kwantyfikatora egzystencjalnego jest następująca:

$\exists$: Identyfikator $\times$ Warunek $\mapsto$ Warunek

$\text{Swa.}[\exists(\text{ide}, \text{wrn})].\text{sta} =$

jest-błąd.sta

$\rightarrow$ *błąd.sta*

niech

$(\text{śro}, (\text{wrt}, 'OK')) = \text{sta}$

istnieje $\text{war} : \text{Wartość}$, $\text{Swa.}[\text{wrn}].(\text{śro}, (\text{wrt}[\text{ide}/\text{war}], 'OK')) = \text{kp} \rightarrow \text{kp}$

dla każdej $\text{war} : \text{Wartość}$, $\text{Swa.}[\text{wrn}].(\text{śro}, (\text{wrt}[\text{ide}/\text{war}], 'OK')) = \text{kf} \rightarrow \text{kf}$

prawda

$\rightarrow$ *'nigdy-prawda'*

Zauważmy, że

$[\forall(\text{ide}, \text{wrn})].\text{sta} = \text{kf}$

nawet wtedy, gdy dla niektórych wartości *war* wartość warunku *wrn* jest komunikatem błędu i analogicznie dla sytuacji, gdy

$[\exists(\text{ide}, \text{wrn})].\text{sta} = \text{kp}$.

Ten wybór oznacza, że kwantyfikatory są definiowane zgodnie z filozofią logiki Kleene'ego, a nie McCarthy'ego. W przypadku wyrażeń boolowskich, które mają być wyliczane w trakcie wykonywania programu, odrzuciłem filozofię Kleene'ego, gdyż prowadziłaby do nieimplementowalnych semantyk. Jednakże w przypadku warunków, o których nie zakładamy, że ich wartości mają być wyliczane przez interpreter, nie tylko możemy sobie na to pozwolić, ale — w przypadku kwantyfikatorów — jest to dla naszych celów wybór lepszy. Może o tym świadczyć przykład warunku:

$(\exists x) (1/x > 2)$

którego wartość w filozofii McCarthy'ego byłaby nieokreślona, gdyż jest nieokreślona dla $x=0$. Więcej na temat konsekwencji tego wyboru w [23] i [50]⁸⁷.

⁸⁷ W tym miejscu może pojawić się pytanie, dlaczego rachunek Kleene'ego przyjąłem jedynie dla kwantyfikatorów, ale już nie dla występujących w warunkach spójników. Spontaniczna odpowiedź jest taka, że skoro klasa warunków zawiera w sobie składniowo klasę wyrażeń boolowskich, to w przypadku tych ostatnich chciałem uniknąć „podwójnej semantyki”, by nie komplikować modelu. Być może jednak należy się jeszcze nad tym wyborem zastanowić.

8.2.2 Warunki danologiczne

Jak już było zapowiedziane, *warunki danologiczne* opisują własności danych języka źródłowego, a ich klasa dzieli się na dwie podklasy:

1. wyrażenia danologiczne języka źródłowego,
2. rozszerzone wyrażenia danologiczne odwołujące się do konstruktorów kompozytów niedostępnych w języku źródłowym.

Warunki drugiej grupy powinny pozwalać na opisywanie tych własności danych, których będziemy chcieli użyć przy opisywaniu własności programów, np. że dana lista jest uporządkowana leksykograficznie, lub że wskazana baza danych spełnia warunki integralności. Zakładam, że semantyka warunków pierwszej grupy pokrywa się z semantyką wyrażeń danologicznych w języku źródłowym.

Jak widzimy, warunki danologiczne nie zawsze przyjmują jako wartości kompozyty boolowskie. Nazywam je jednak „warunkami”, bo tworzą podzbiór kategorii Warunek.

8.2.3 Warunki walidacyjne

Warunki walidacyjne opisują własności stanów związane z kontekstem programistycznym, są więc specyficzne dla programowania walidującego, choć najczęściej uniwersalne z punktu widzenia źródłowej algebry danych. W niniejszym rozdziale podam kilka przykładów klas warunków walidacyjnych, które — jak sądzę — powinny się znaleźć w każdym języku programowania walidacyjnego. W odróżnieniu od warunków danologicznych, warunki walidacyjne zawsze przyjmują jako wartości kompozyty boolowskie (lub błędy).

Na początek zdefiniuję dwa warunki o stałych wartościach logicznych, które na poziomie składni konkretnej oznaczam przez **TT** i **FF**. Tak więc

[**TT**].sta =

jest-błąd.sta → błąd.sta

prawda → kp

[**FF**].sta =

jest-błąd.sta → błąd.sta

prawda → kf

Zakładam też, że dla każdego wyrażenia danologicznego *wyd* istnieje w języku warunek, który zapisujemy jako **defined-d**(*wyd*), mówiący, że w danym stanie wartość wyrażenia *wyd* jest określona:

[**defined-d**(*wyd*)].sta =

jest-błąd.sta → błąd.sta

Swd.[*wyd*].sta = ? → kf

Swd.[*wyd*].sta : Błąd → kf

prawda → kp

Zauważmy, że ponieważ wyrażenia danologiczne obejmują wywołania procedur funkcyjnych, warunek **defined**(*wyr*) może być nieobliczalny, co jednak nie stanowi problemu, ponieważ nie będzie on występował w warstwie programistycznej języka.

Analogiczne warunki tworzymy dla wyrażeń typologicznych, z tym że tym razem nie musimy sprawdzać określoności.

```
[defined-t (wyt)].sta =
  jest-błąd.sta      → błąd.sta
  Swty.[wyt].sta : Błąd → kf
  prawda             → kp
```

Wśród warunków walidacyjnych znajdują się też warunki mówiące, że wskazany identyfikator jest zadeklarowanym identyfikatorem zmiennej określonego typu:

```
[ide is wyt].sta =
  jest-błąd.sta      → błąd.sta
  niech
    (śro, (wrt, 'OK')) = sta
  wrt.ide = ?        → kf
  Swty.[wyt].sta : Błąd → Swty.[wyt].sta
  niech
    (dan, typ) = Swd.[ide].sta
    typ-w      = Swty.[wyt].sta
  typ = typ-w       → kp
  typ ≠ typ-w       → kf
```

Przykładem takiego warunku może być:

```
waga is real
```

lub też

```
pracownik is record-type
  imię          as word,
  nazwisko      as word,
  rok-urodzenia as number,
  lata-nagród   as array-of number ee
ee
```

Poniżej opisuję kolejne trzy klasy warunków niezbędne w każdym języku programowania walidującego.

Warunki pierwszej klasy wyrażają fakt zgodności typu danego identyfikatora z typem wartości danego wyrażenia.

```
[ide conformant-with wyd].sta = (typ ide jest zgodny z typem wartości wyd)
  jest-błąd.sta      → błąd.sta
  niech
    sta = (śro, wrt, 'OK')
```

wrt.ide = ? → kf
Swd.[w_yd].sta = ? → kf
Swd.[w_yd].sta : Błęd → Swd.[w_yd].sta

niech

(dan-i, (kor-i, jar-i) = wrt.ide

(dan-w, kor-w) = Swd.[wyd]

kor-i = kor-w → kp

prawda → kf

Zatem identyfikator i wyrażenie są zgodne, gdy identyfikator jest zadeklarowany, a wartość wyrażenia jest określona i jej korpus jest zgodny z korpusem wartości identyfikatora. W tej definicji przyjąłem, że nieokreśloność wartości wyrażenia `wyrd`, która może mieć miejsce jedynie w przypadku niekończącego się obliczenia, powoduje że warunek przyjmuje wartość `kf`. To oczywiście może uczynić ten warunek nieobliczalnym.

Zauważmy też, że w odróżnieniu od poprzednich warunków, w którym oczekujemy zgodności typów, w tym przypadku ograniczamy się do zgodności korpusów, bowiem wynikiem wykonania wyrażenia jest kompozyt, a nie wartość.

Druga klasa warunków odpowiada zdefiniowanej w rozdziale 7.2.2 funkcji dynamicznie-zgodnej dotyczącej zgodności parametrów aktualnych wywołania procedury z jej parametrami formalnymi. Niech pfo-w, pfo-r, pak-w, pak-r będą listami odpowiednio parametrów formalnych wartościowych i referencyjnych oraz parametrów aktualnych wartościowych i referencyjnych:

```
[conformant(pfo-w, pfo-r, pak-w, pak-r)].sta =
```

jest-blad.sta → blad.sta

niech

 $((\acute{s}rt, \acute{s}rp), (wrt, 'OK')) = sta$

dynamicznie-zgodne.(pfo-w, pfo-r, pak-w, pak-r).(śrt, wrt) = 'OK' → kp

prawda → kf

Zdefiniowanie trzeciej klasy warunków wymaga wprowadzenia pewnych pojęć pomocniczych. Stan nazwę *inicjalnym* jeżeli jest postaci

$$(([], []), ([], \text{'OK'}))$$

gdzie $[\]$ oznacza funkcję pusta, a więc wszędzie nieokreślona.

Stan nazwę *adekwatnym* względem danej preambuły `pab`, jeżeli nie niesie on komunikatu błędu oraz powstaje ze stanu inicjalnego przez wykonanie dowolnego programu, którego preambuła jest identyczna z `pab`. Innymi słowy stan jest adekwatny względem `pab` jeżeli:

1. nie niesie komunikatu błędu,
2. w środowisku procedur tego stanu są zadeklarowane wszystkie i jedynie te procedury, których deklaracje znajdują się w preambule,
3. w środowisku typów tego stanu są zdefiniowane wszystkie i jedynie te typy, których definicje znajdują się w preambule,

4. w wartościowaniu tego stanu są zadeklarowane wszystkie i jedynie te zmienne, których deklaracje znajdują się w preambule,
5. typy zadeklarowanych zmiennych zostały zdefiniowane w preambule⁸⁸.

Jak stąd wynika, wszystkie stany, które pojawiają się w trakcie wykonania jakiegokolwiek zawartej w programie instrukcji i nie niosą błędu, są adekwatne względem preambuły tego programu.

Przez $AD.pab$ oznaczam zbiór wszystkich stanów adekwatnych względem preambuły pab . Warunki zwykle trzeciej grupy są postaci **ade-for** (pab) gdzie pab jest preambuła i mają niżej określoną semantykę:

$[ade-for(pab)].sta =$
 $jest-błąd.sta \rightarrow błąd.sta$
 $sta : AD.pab \rightarrow kp$
 $prawda \rightarrow kf.$

8.2.4 Warunki algorytmiczne

Warunki algorytmiczne⁸⁹ są składniowo postaci

$ins @ wrn$

gdzie ins jest dowolną instrukcją, a wrn — dowolnym warunkiem, a w tym być może warunkiem algorytmicznym. Semantyka warunków algorytmicznych jest zdefiniowana równaniem

$Swa.[ins @ wrn] = Sin.[ins] \bullet Swa.[wrn]$

Tak więc wartość warunku $ins @ wrn$ w stanie sta to wartość warunku wrn w stanie $Sin.[ins].sta$ będącym wynikiem wykonania instrukcji ins w stanie początkowym sta . Jak wynika z rozważań prowadzonych w rozdziale 3.4, $ins @ wrn$ to najsłabszy prewarunek gwarantujący, że instrukcja ins się wykona, a stan końcowy spełni warunek wrn .

O warunku, który nie jest algorytmicznym powiemy, że jest w *postaci standardowej*. Warunki algorytmiczne, podobnie jak danologiczne, mogą przyjmować kompozyty nieboolowskie jako swoje wartości.

8.3 Instrukcje specyfikowane

Jak już było zapowiedziane *instrukcje specyfikowane*, zwane w skrócie *specinstrukcjami*, można sobie wyobrazić jako instrukcje z zagnieżdżonymi w nich warunkami wyrażającymi własności stanów pojawiających się w trakcie wykonywania instrukcji. Definiująca je klauzula gramatyczna jest następująca:

$sin : SpecInstrukcja =$
 $Instrukcja \quad |$

⁸⁸ Ten warunek jest w zasadzie niepotrzebny, bo jest konsekwencją 1. i 4., umieszczam go jednak, aby nie został przeoczony.

⁸⁹ Warunki tego typu legły u podstaw tzw. *logiki algorytmicznej* intensywnie rozwijanej na Uniwersytecie Warszawskim w latach 1970-1980 (patrz [8]).

```

asr Warunek rsa |
if WyrDan then SpecInstrukcja else SpecInstrukcja fi |
if-error WyrDan then SpecInstrukcja fi |
while WyrDan do SpecInstrukcja od |
SpecInstrukcja ; SpecInstrukcja

```

Powyższe równanie rozszerza gramatykę **Lingua-2** o nową klauzulę, a sam język o nowy rodzaj. Jak widzimy, specinstrukcje zawierają w sobie wszystkie instrukcje oraz jedną konstrukcję dodatkową **asr wrn rsa**, którą nazwiemy *asercją*. Pozostałe cztery klauzule pozwalają budować specinstrukcje strukturalne podobnie jak ma to miejsce w przypadku instrukcji.

Denotacje specinstrukcji są, podobnie jak denotacje instrukcji, funkcjami częściowymi na stanach, a zatem ich semantyka jest funkcją:

$$\text{Ssi} : \text{SpecInstrukcja} \mapsto \text{Stan} \rightarrow \text{Stan}$$

Ta funkcja jest zdefiniowana w następujący sposób:

$$\text{Ssi}[\text{ins}] = \text{Sin}[\text{ins}]$$

$$\text{Ssi}[\text{off ins on}] = \text{Sin}[\text{ins}]$$

$$\text{Ssi}[\text{asr wrn rsa}].\text{sta} =$$

$$\text{jest-błąd.sta} \rightarrow \text{sta}$$

$$\text{Swa}[\text{wrn}].\text{sta} = ? \rightarrow ?$$

$$\text{Swa}[\text{wrn}].\text{sta} = \text{kp} \rightarrow \text{sta}$$

$$\text{prawda} \rightarrow \text{sta} \blacktriangleleft \text{'niespełniona-asercja'}$$

Semantyka specinstrukcji będących instrukcjami pokrywa się z semantyką instrukcji, natomiast w przypadku asercji, jeżeli stojący w asercji warunek jest spełniony, to stan nie ulega zmianie, a w przeciwnym przypadku pojawia się komunikat błędu. Zauważmy, że komunikat błędu 'niespełniona-asercja' pojawi się w dwóch przypadkach:

1. gdy wartością warunku będzie (ff, ('boolowska')),
2. gdy wartością warunku będzie błąd.

Dla pozostałych czterech klauzul gramatycznych nasza semantyka jest zdefiniowana analogicznie jak dla instrukcji, nie będę więc tych definicji przytaczał.

Jak się przekonamy w dalszych rozdziałach, opisana powyżej składnia i semantyka specinstrukcji stanowi fundament do definiowania reguł budowania metaprogramów poprawnych. W tym kontekście instrukcje asercji służą do opisywania własności stanów pojawiających się pomiędzy sąsiadującymi ze sobą *instrukcjami atomowymi* — przypisaniami i wywołaniami procedur — i są wykorzystywane w regułach transformowania metaprogramów z zachowaniem ich poprawności.

Często jest tak, że jakaś asercja jest spełniona na obszarze zawierającym pewną liczbę wykonywanych sukcesywnie instrukcji, być może z wyłączeniem jakiegoś ich podobszaru. W takich przypadkach, aby uniknąć wielokrotnego powtarzania tej samej asercji w wielu miejscach programu, będziemy się posługiwali dwoma skrótami składniowymi postaci:

begin-asr wrn ; sin **end-asr** (*)

off-asr sin **on-asr** (**)

Pierwszy z nich jest kolokwializmem odpowiadającym specinstrukcji powstającej z **sin** przez wpisanie do niej asercji **asr wrn rsa** pomiędzy każde dwie instrukcje atomowe za wyjątkiem obszaru instrukcji objętej wyłączeniem asercji **off-asr on-asr** oraz instrukcji obsługi błędu. Występującą w (*) specinstrukcję będziemy nazywać *obszarem obowiązywania warunku* pamiętając, że w tym obszarze mogą się pojawić wyłączenia. Mówimy też, że na obszarze instrukcji **sin** zadekretowano obowiązywanie warunku **wrn**.

Skrót (**) jest również kolokwializmem wskazującym intuicyjnie, że na obszarze instrukcji **sin** zadekretowano wyłączenie wszystkich zadekretowanych wcześniej warunków⁹⁰.

Tak więc, ani (*) ani (**) nie są specinstrukcjami, a jedynie skrótami notacyjnymi, które zostaną formalnie zdefiniowane jako kolokwializmy, co oznacza, że nie wystąpią one na poziomie składni konkretnej, a więc też i na poziomie denotacji. Wybieram tę drogę, gdyż semantyki dla (*) nie da się opisać w sposób denotacyjny. Gdyby bowiem tak miało być, to denotacja każdej konstrukcji postaci

Ssi.[**begin-asr wrn; sin end-asr**]

musiałaby dać się wyrazić jako złożenie denotacji jej składowych, a więc jako złożenie **Swa.[wrn]** oraz **Ssi.[sin]**. To jednak nie jest możliwe, o czym świadczy poniższy przykład dwóch specinstrukcji, które w naszym zamierzeniu powinny mieć różne denotacje, mimo że denotacje ich składowych są identyczne:

begin-asr x > 0;

x := x

end-asr

begin-asr x > 0;

x := -x ; x := -x

end-asr

W związku z tym (*) i (**) są definiowane jako kolokwializmy opisane odpowiednią transformacją przywracającą TP. Ta transformacja będzie identycznościowa dla specinstrukcji, w których nie występują obszary obowiązywania warunków, natomiast gdy tak nie jest TP zostanie zdefiniowana indukcyjnie względem struktury specinstrukcji, którą obejmuje.

Rozpocniemy do przypadku, gdy zawarta w obszarze obowiązywania warunku specinstrukcja jest zwykłą instrukcją, a ten przypadek rozpocniemy od podprzypadku przypisania:

TP.[**begin-asr wrn; ide := wyd end-asr**] =

asr wrn rsa; ide:= wyd; asr wrn rsa

⁹⁰ Dla uproszczenia naszego modelu przyjmuję, że wyłączenia powodują wyłączenie wszystkich warunków o być może zagnieżdżonych obszarach obowiązywania. Alternatywą jest oczywiście wprowadzenie wyłączeń skazujących wyłączany warunek, tę jednak wersję pozostawiam na razie do przyszłych rozważań.

Dla instrukcji przypisania jarzma, i wywołania procedury, naszą transformację definiujemy analogicznie. Kolejny przypadek to instrukcja obsługi błędu. W tym przypadku zakładam, że taka instrukcja jest wyłączona z obszaru obowiązywania warunku:

```
TP.[begin-asr wrn; if-error wyd then ins fi end-asr] =
  asr wrn rsa; if-error wyd then ins fi asr wrn rsa
```

Dalsze podprzypadki, gdy obszarem obowiązywania warunku jest objęta instrukcja (a nie specinstrukcja) definiujemy w następujący sposób:

```
TP.[begin-asr wrn; if wyd then ins-1 else ins-2 fi end-asr] =
  asr wrn rsa;
  if wyd
    then TP.[begin-asr wrn; ins-1 end-asr]
    else TP.[begin-asr wrn; ins-2 end-asr]
  fi;
  asr wrn rsa
```

```
TP.[begin-asr wrn; while wyd do ins od end-asr] =
  asr wrn rsa;
  while wyd do TP.[begin-asr wrn; ins end-asr] od;
  asr wrn rsa
```

```
TP.[begin-asr wrn; ins-1 ; ins-2 end-asr] =
  asr wrn rsa;
  TP.[begin-asr wrn; ins-1 end-asr];
  asr wrn rsa;
  TP.[begin-asr wrn; ins-2 end-asr]
  asr wrn rsa
```

```
TP.[begin-asr wrn; if-error wyd then ins fi end-asr] =
  begin-asr wrn end-asr;
  if-error wyd then TP.[begin-asr wrn; ins end-asr] fi;
  begin-asr wrn end-asr;
```

Teraz należy rozważyć przypadki, gdy obszarem obowiązywania warunku jest objęta specinstrukcja, która nie jest instrukcją.

```
TP.[begin-asr wrn off ins on end-asr] = ins
```

Jak widzimy asercja nie „wnika” do bloku ograniczonego wyłączeniem.

```
TP.[begin-asr wrn-1; asr wrn-2 rsa end-asr] =
  asr wrn-1 and wrn-2 rsa
```

```
TP.[begin-asr wrn-1; begin-asr wrn-2; sin end-asr end-asr] =
  TP.[begin-asr wrn-1 and wrn-2; sin end-asr]
```

Pozostałe przypadki specinstrukcji strukturalnych definiują się analogicznie jak odpowiadające im przypadki dla instrukcji zwykłych.

8.4 Tezy

Najogólniej rzecz ujmując warunki służą do wyrażania własności stanów, a tezy do wyrażania własności warunków, specinstrukcji i programów, a w tym również składniowych własności tych ostatnich. W naszym języku mamy do czynienia z trzema rodzajami tez:

- *właściwości składniowe* — programów i ich składowych,
- *metawarunki* — semantyczne własności warunków,
- *metaprogramy* — semantyczne własności instrukcji specyfikowanych.

W odróżnieniu od warunków, które jako wartości przyjmują kompozyty, i to niekoniecznie boolowskie, a także błędy, i które dla pewnych stanów mogą być nieokreślone, wartościami tez są zawsze określone i są nimi klasyczne wartości logiczne pp i ff. O ile bowiem w programach korzystamy z predykatów częściowych i wielowartościowych, o tyle w opisach tych programów pozostajemy na gruncie logiki klasycznej.

Kategorię tez ponownie — czyli jak w przypadku warunków i z tych samych powodów — buduję w porządku odwrotnym niż zalecany w niniejszej książce, a mianowicie od składni do denotacji.

8.4.1 Właściwości składniowe

Właściwościami składniowymi nazywam tezy wyrażające syntaktyczne własności programów i ich składowych. Dla zdefiniowania podstawowych właściwości kilka pojęć i oznaczeń pomocniczych.

Jeżeli pab jest preambułą, to przez names.pab oznaczam zbiór wszystkich identyfikatorów procedur, funkcji, zmiennych i typów zadeklarowanych w pab. Ponieważ pojedyncze

deklaracje każdego z rodzajów można traktować jako preambuły, również do nich stosuje się funkcja `names`.

O dwóch preambułach powiemy, że są *rozdzielne*, gdy zbiory deklarowanych w nich zmiennych są rozłączne.

Powiemy, że preambuła `pab` jest *poprawna*, jeżeli żaden identyfikator nie został w niej zadeklarowany dwukrotnie. Zauważmy, że preambuła jest poprawna wtedy i tylko wtedy, gdy zbiór jej stanów adekwatnych (rozdział 8.2.2) jest niepusty. Powiemy, że deklaracja lub definicja jest *dopuszczalna* w preambule, jeżeli jej dodanie nie spowoduje, że preambuła stanie się niepoprawna.

Wprowadzam niżej określone konstruktory właściwości składniowych:

<code>is-correct pab</code>	— preambuła <code>pab</code> jest poprawna
<code>dec is-in pab</code>	— deklaracja <code>dec</code> występuje w preambule <code>pab</code>
<code>dec allowed-in pab</code>	— deklaracja <code>dec</code> jest dopuszczalna w <code>pab</code>
<code>ide is-pro-in pab</code>	— <code>ide</code> jest zadeklarowany w <code>pab</code> jako procedura
<code>ide is-fun-in pab</code>	— <code>ide</code> jest zadeklarowany w <code>pab</code> jako funkcja
<code>ide is-typ-in pab</code>	— <code>ide</code> jest zdefiniowany w <code>pab</code> jako typ
<code>ide is wyt in pab</code>	— <code>ide</code> jest zadeklarowany w <code>pab</code> jako zm. typu <code>wyt</code>
<code>ide not-in pab</code>	— <code>ide</code> nie jest zadeklarowany w <code>pab</code> w żaden sposób
<code>pab-1 separated-from pab-2</code>	— <code>pab-1</code> i <code>pab-2</code> są rozdzielne

Definicje formalne pomijam jako oczywiste.

8.4.2 Metawarunki

Metawarunki opisują własności warunków. Dla ich zdefiniowania wprowadzam notację, która okaże się wygodna w dalszych rozważaniach.

$$[\text{wrn}] = \text{Swa}.[\text{wrn}]^{91}$$

$$\{\text{wrn}\} = \{\text{sta} : \text{Swa}[\text{wrn}].\text{sta} = \text{kp}\}$$

gdzie `wrn` jest dowolnym warunkiem. *Metawarunki* tworzymy z warunków za pomocą czterech konstruktorów, które nazywam *metapredykatami*:

$$\Rightarrow, \sqsubseteq, \Leftrightarrow, \equiv : \text{Warunek} \times \text{Warunek} \mapsto \text{Teza}$$

Denotacjami metawarunków są klasyczne wartości logiczne `pp` lub `ff`, natomiast metapredykaty odpowiadają relacjom binarnym pomiędzy warunkami. Metapredykaty definiuję w sposób następujący⁹²:

$$\begin{array}{llll} \text{wrn-1} \Rightarrow \text{wrn-2} & \mathbf{wtw} & \{\text{wrn-1}\} \subset \{\text{wrn-2}\} & (\text{silniejszy niż}) \\ \text{wrn-1} \sqsubseteq \text{wrn-2} & \mathbf{wtw} & [\text{wrn-1}] \subset [\text{wrn-2}] & (\text{slabiej określony niż}) \end{array}$$

⁹¹ W tym miejscu wprowadzam redundantne oznaczenie na semantykę warunków, które jednak okaże się wygodne przy dalszych rozważaniach o warunkach.

⁹² „**wtw**” to skrót od „wtedy i tylko wtedy gdy”.

$wrn-1 \Leftrightarrow wrn-2$ **wtw** $\{wrn-1\}=\{wrn-2\}$ (*słabo równoważne*)

$wrn-1 \equiv wrn-2$ **wtw** $[wrn-1]=[wrn-2]$ (*silnie równoważne*)

W pierwszym przypadku będziemy również mówili również, że $wrn-2$ *jest słabszy niż* $wrn-1$, a w drugim — że $wrn-2$ *jest silniej określony niż* $wrn-1$. Pomiędzy metapredykatami zachodzą dość oczywiste związki:

$wrn-1 \equiv wrn-2$ implikuje $wrn-1 \Leftrightarrow wrn-2$

$wrn-1 \equiv wrn-2$ implikuje $wrn-1 \sqsubseteq wrn-2$

$wrn-1 \equiv wrn-2$ jest równoważne ($wrn-1 \sqsubseteq wrn-2$ oraz $wrn-2 \sqsubseteq wrn-1$)

$wrn-1 \Leftrightarrow wrn-2$ jest równoważne ($wrn-1 \Rightarrow wrn-2$ oraz $wrn-2 \Rightarrow wrn-1$)

$wrn-1 \Leftrightarrow wrn-2$ implikuje $wrn-1 \Rightarrow wrn-2$

Przy pomocy metapredykatów można w prosty sposób wyrazić własność częściowej i całkowitej poprawności instrukcji ins względem prewarunku pre i postwarunku $post$:

$pre @ ins \Rightarrow post$ — poprawność częściowa

$pre \Rightarrow ins @ post$ — poprawność całkowita

a także opisać własności *słabego* i *mocnego niezmiennika* instrukcji:

$wrn @ ins \Rightarrow wrn$ — słaby niezmiennik

$wrn \Rightarrow ins @ wrn$ — mocny niezmiennik

Słabych i mocnych niezmienników używamy w dowodach odpowiednio częściowej i całkowitej poprawności programów. Przyjrzyjmy się teraz kilku przykładom⁹³:

$x > 0$ **and** $\sqrt[2]{x} > 2$ $\equiv$ $x > 4$

$\sqrt[2]{x} > 2$ $\Leftrightarrow$ $x > 4$ jednak $\equiv$ nie zachodzi

$\sqrt[2]{x} < 2$ $\sqsubseteq$ $x < 4$ jednak ani $\equiv$ ani $\Leftrightarrow$ nie zachodzą

$\sqrt[2]{x} > 4$ $\Rightarrow$ $x > 3$ jednak ani $\Leftrightarrow$ ani $\sqsubseteq$ nie zachodzą

Zauważmy też, że⁹⁴:

$wrn-1 \Rightarrow wrn-2$ nie pociąga za sobą, że $(wrn-1 \text{ **implies** } wrn-2) \equiv \mathbf{TT}$.

Rzeczywiście mimo że $\sqrt[2]{x} > 4 \Rightarrow x > 3$, warunek

$\sqrt[2]{x} > 4 \text{ **implies** } x > 3$

jest nieokreślony dla $x < 0$. Jednakże wynikanie w drugą stronę ma miejsce:

jeżeli $(wrn-1 \text{ **implies** } wrn-2) \equiv \mathbf{TT}$, to $wrn-1 \Rightarrow wrn-2$.

Rzeczywiście, niech $sta:\{wrn-1\}$, co oznacza, że $[wrn-1].sta = kp$. Jeżeli teraz $[(wrn-1 \text{ **implies** } wrn-2)].sta = kp$ oraz $[wrn-1].sta = kp$, to musi zachodzić $[wrn-2].sta = kp$, czyli $sta:\{wrn-2\}$.

⁹³ Zakładam, że pierwiastek kwadratowy z liczby ujemnej nie jest określony.

⁹⁴ Implikację definiujemy w klasyczny sposób tj. $p \text{ **implies** } q$ oznacza $(\text{not } p) \text{ or } q$, gdzie negacja i alternatywa są McCarthy'owskie.

Zauważmy, że na gruncie naszego nieklasycznego rachunku warunków możemy mówić o dwóch rodzajach spełnialności warunku:

$wrn \equiv \mathbf{TT}$ — *silna spełnialność*; wrn jest zawsze prawdziwy

$wrn \sqsubseteq \mathbf{TT}$ — *słaba spełnialność*; wrn nigdy nie jest fałszywy

Zauważmy teraz, że definiowane powyżej metapredykaty mogą być potraktowane jako relacje binarne w algebrze warunków. Łatwo udowodnić następujące lematy (więcej w [21]):

Lemat 8.4.2-1 *Relacje $\equiv$ oraz $\Leftrightarrow$ są równoważnościami w zbiorze warunków, tj. są zwrotne, symetryczne i przechodnie. ■*

Lemat 8.4.2-2 *Silna równoważność jest kongruencją, tj. zastąpienie w dowolnym warunku jego podwarunków warunkami silnie równoważnymi daje w wyniku warunek silnie równoważny z wyjściowym. ■*

Lemat 8.4.2-3 *Słaba równoważność jest kongruencją jedynie względem **and** oraz **or**. ■*

Słaba równoważność nie jest kongruencją względem negacji, tzn.

$$wrn-1 \Leftrightarrow wrn-2 \text{ nie implikuje } \mathbf{not} \ wrn-1 \Leftrightarrow \mathbf{not} \ wrn-2$$

Na przykład, jest prawdą

$$\sqrt[2]{x} > 2 \Leftrightarrow x > 4$$

ale nie jest prawdą

$$\sqrt[2]{x} \leq 2 \Leftrightarrow x \leq 4$$

gdyż dla $x=-1$ wyrażenie po prawej stronie przyjmuje wartość kp, podczas gdy po lewej generuje komunikat błędu.

Lemat 8.4.2-4 *Spójniki **and** oraz **or** są silnie łączne, tzn.*

$$(wrn-1 \ \mathbf{and} \ wrn-2) \ \mathbf{and} \ wrn-3 \equiv wrn-1 \ \mathbf{and} \ (wrn-2 \ \mathbf{and} \ wrn-3)$$

$$(wrn-1 \ \mathbf{or} \ wrn-2) \ \mathbf{or} \ wrn-3 \equiv wrn-1 \ \mathbf{or} \ (wrn-2 \ \mathbf{or} \ wrn-3) \quad \blacksquare$$

Oczywiście są również słabo łączne, gdyż silna równoważność implikuje słabą.

Lemat 8.4.2-5 *Spójnik **and** jest silnie lewostronnie rozdzielny względem **or** i na odwrót, tzn.*

$$wrn-1 \ \mathbf{and} \ (wrn-2 \ \mathbf{or} \ wrn-3) \equiv$$

$$wrn-1 \ \mathbf{and} \ wrn-2) \ \mathbf{or} \ (wrn-1 \ \mathbf{and} \ wrn-3)$$

$$wrn-1 \ \mathbf{or} \ (wrn-2 \ \mathbf{and} \ wrn-3) \equiv$$

$$(wrn-1 \ \mathbf{or} \ wrn-2) \ \mathbf{and} \ (wrn-1 \ \mathbf{or} \ wrn-3) \quad \blacksquare$$

Jednakże oba spójniki nie są silnie prawostronnie rozdzielne. Rzeczywiście:

$$\begin{aligned} (kp \text{ or } bb) \text{ and } kf &= kf \quad \text{ale} \quad (kp \text{ and } kf) \text{ or } (bb \text{ and } kf) = bb \\ (kf \text{ and } bb) \text{ or } kp &= kp \quad \text{ale} \quad (kf \text{ or } kp) \text{ and } (bb \text{ or } kp) = bb \end{aligned} \quad (8.4.2-1)$$

Lemat 8.4.2-6 Spójnik **and** jest słabo lewostronnie rozdzielny względem **or** tzn.

$$\begin{aligned} (wrn-1 \text{ or } wrn-2) \text{ and } wrn-3 &\Leftrightarrow \\ &(wrn-1 \text{ and } wrn-3) \text{ or } (wrn-2 \text{ and } wrn-3) \quad \blacksquare \end{aligned}$$

Jednakże **or** nie jest nawet słabo lewostronnie rozdzielne względem **and** o czym świadczy przykład (8.4.2-1).

Lemat 8.4.2-7 Prawa de Morgana dla **and** oraz **or** i prawa negacji kwantyfikatorów są spełnione dla silnej równoważności ■

Lemat 8.4.2-8 Koniunkcja jest słabo przemienna, tzn.

$$wrn-1 \text{ and } wrn-2 \Leftrightarrow wrn-2 \text{ and } wrn-1 \quad \blacksquare$$

Jednakże koniunkcja nie jest silnie przemienna, a alternatywa nie jest nawet słabo przemienna, gdyż:

$$kp \text{ or } bb = kp \quad \text{ale} \quad bb \text{ or } kp = bb$$

Lemat 8.4.2-9 Jeżeli

$$wrn-1 \Rightarrow wrn-2$$

to

$$wrn-1 \text{ and } wrn-2 \equiv wrn-1 \quad \blacksquare$$

Obok metapredykatów dwuargumentowych definiujemy też metapredykaty trójargumentowe, które znajdują zastosowanie przy konstruowaniu programów poprawnych:

$$wrn-1 \equiv wrn-2 \text{ whenever } wrn \text{ wtw } wrn \text{ and } wrn-1 \equiv wrn \text{ and } wrn-2$$

$$wrn-1 \Leftrightarrow wrn-2 \text{ whenever } wrn \text{ wtw } wrn \text{ and } wrn-1 \Leftrightarrow wrn \text{ and } wrn-2$$

W obu tych przypadkach powiemy, że wrn stanowi *kontekst logiczny* lub po prostu *kontekst* dla równoważności po której następuje. Będziemy też mówili, że *równoważność* $wrn-1 \equiv wrn-2$ *jest spełniona w kontekście* wrn lub *pod warunkiem*, że wrn i analogicznie dla słabej równoważności. A oto dwa przykłady:

$$n > x^2 \equiv \sqrt[3]{n} > x \text{ whenever } (n \geq 0 \text{ and } x \geq 0)$$

$$n > x^2 \Leftrightarrow \sqrt[3]{n} > x \text{ whenever } x \geq 0$$

Tym kontekstem będzie najczęściej warunek w obszarze dekracji którego będziemy chcieli dokonać zamiany jednego warunku na drugi.

Materiał przedstawiony w niniejszym rozdziale został opublikowany w latach 1980. w moich pracach [19] i [23], a rozwinięcie tych idei w kierunku trójwartościowych teorii dedukcyjnych w pracy [50] napisanej wspólnie z Beatą Konikowską i Andrzejem Tarleckim.

8.4.3 Metaprogramy

Metaprogramy są tezami, których składnia konkretna jest określona poniższym równaniem (dla uproszczenia poniższych zapisów pomijam nawiasy obejmujące programy, tj. **begin-program** oraz **end-program** wprowadzone w rozdziale 6.2.2):

```
mep : MetaProgram =
  def Preambuła
  pre Warunek
    SpecInstrukcja
  post Warunek
```

Wyrażają one całkowitą poprawność specinstrukcji (w sensie opisanym w rozdziale 3.6) zrelatywizowaną do stanów adekwatnych dla preambuły. Semantyka metaprogramów jest więc funkcją typu:

$$\text{Smp} : \text{MetaProgram} \mapsto \{\text{pp}, \text{ff}\}$$

a definiujemy ją w sposób następujący:

$$\text{Smp}[\text{def pab pre prw sin post pow}] = \text{pp}$$

wtedy i tylko wtedy gdy preambuła **pab** jest poprawna oraz

$$\{\text{ade-for}(\text{pab}) \text{ and prw} \} \subset \text{Ssi}[\text{sin}] \bullet \{\text{pow}\} \quad (8.4-1)$$

czyli

$$\text{ade-for}(\text{pab}) \text{ and prw} \Rightarrow \text{sin} @ \text{pow}$$

Gdy denotacją metaprogramu jest **pp**, to mówimy, że ten metaprogram jest *poprawny*.

Tak więc metaprogram jest poprawny, gdy poprawna jest jego preambuła oraz dla każdego stanu adekwatnego względem tej preambuły, jeżeli ten stan spełnia prewarunek, to wykonanie w tym stanie zawartej w programie instrukcji zakończy się pomyślnie i stan końcowy spełni postwarunek.

Zauważmy, że pomyślne zakończenie **sin** oznacza m.in. że żaden ze stanów pośrednich nie naruszył zadeklarowanych asercji oraz że stan końcowy nie niesie błędu, gdyż wtedy **post-warunek** nie byłby spełniony.

Dla tak określonej poprawności metaprogramów możemy sformułować kilka użytecznych lematów. Poniższy wynika wprost z właśnie sformułowanej uwagi.

Lemat 8.4.3-1 *Jeżeli*

```
def pab pre wrn-pr sin post wrn-po
```

jest poprawny, a sin-1 powstało z sin przez usunięcie z niego dowolnej liczby asercji lub deklaracji asercji, to program

```
def pab pre wrn-pr sin-1 post wrn-po
```

jest również poprawny. ■

Lemat 8.4.3-2 *Jeżeli*

def pab **pre** wrn-pr **sin** **post** wrn-po

jest poprawny, to poprawny będzie również metaprogram powstający z niego przez zastąpienie dowolnego z występujących w nim warunków przez warunek z nim słabo równoważny. ■

Dowód wynika z faktu, że denotacje asercji, których warunki są słabo równoważne, są identyfikatorami na wspólnym zbiorze stanów. Z tego lematu wynika w szczególności, że na poziomie warunków (ale nie wyrażeń boolowskich w warstwie programistycznej!) możemy korzystać ze wszystkich lematów rozdziału 8.4.2 dotyczących słabych równoważności.

Lemat 8.4.3-3 *Jeżeli*

def pab **pre** wrn-pr **sin** **post** wrn-po

*jest poprawny, to poprawny będzie również metaprogram powstający przez zastąpienie w nim dowolnego wyrażenia danologicznego o wartościach boolowskich `wyd` stojącego w rozgałęzieniu **if-then-else-fi** lub **while-do-od** wyrażeniem `wyd-1` silniej określonym, tj. takim że $wyd \sqsubseteq wyd-1$, a w szczególności gdy $wyd \equiv wyd-1$* ■

Jeżeli program wyjściowy jest poprawny, to w żadnym przebiegu tego programu żadne wyrażenie boolowskie nie może generować sygnału błędu, a wszędzie tam, gdzie `wyd` przyjmuje wartość określoną `wyd-1` przyjmuje tę samą wartość.

Zauważmy teraz, że dla dowolnej preambuły pab warunek **ade-for** (pab) jest słabym niezmiennikiem każdej specinstrukcji⁹⁵, gdyż specinstrukcje nie zmieniają ani środowiska ani typów zmiennych globalnych. Oznacza to, że warunek (8.4-1) jest równoważny warunkowi

$$\{\mathbf{ade-for}(pab) \text{ and } prw\} \subset \mathbf{Sin}.[\mathbf{sin}] \bullet \{\mathbf{ade-for}(pab) \text{ and } pow\}$$

Mamy więc do czynienia ze “zwykłą” całkowitą poprawnością, tyle że w zbiorze stanów adekwatnych dla preambuły. To właśnie miałem na myśli pisząc wcześniej, że poprawność metaprogramu jest zrelatywizowana do jego preambuły⁹⁶. W dalszych rozważaniach będę przyjmował, że:

def pab **pre** wrn-pr **sin** **post** wrn-po

wyraża tezę, że dany metaprogram jest metaprogramem poprawnym (skr. jest MPP). Postępuję w tym przypadku podobnie jak w codziennej matematyce, gdzie pisząc „ $x > 2$ ” mamy na myśli, że „ $x > 2$ jest prawdą”.

⁹⁵ Wyciągnięcie tego wniosku wymaga skorzystania z faktu, że denotacjami specinstrukcji są funkcje, tj. że specinstrukcje mają charakter deterministyczny, gdyż jedynie wtedy całkowita poprawność implikuje częściową.

⁹⁶ To dla uzyskania tej właściwości metaprogramów przyjąłem, że wszystkie deklaracje globalne w programie poprzedzają wszystkie jego instrukcje.

8.4.4 Paradoks de Bakker'a w logice Hoare'a

Jak zauważył Jaco de Bakker (str. 108, rozdział 4 w [5]), a następnie skomentował Krzysztof Apt (str. 460 rozdział 5 w [4]), na gruncie logiki Hoare'a daje się udowodnić formułę:

$$\{ \text{true} \} a[a[2]] := 1 \{ a[a[2]] = 1 \}$$

która nie dla każdej tablicy a będzie prawdziwa. Rzeczywiście, jeżeli założymy, że

$$a = [2, 2]$$

to

$$a[2] = 2$$

a stąd wykonanie przypisania

$$a[a[2]] := 1$$

oznacza wykonanie przypisania

$$a[2] := 1$$

a więc nowa tablica to $a = [2, 1]$, skąd $a[a[2]] = a[1] = 2$.

Zauważmy jednak, że ten problem nie bierze się ani z wprowadzenia do języka tablic, ani z możliwości utworzenia wyrażenia typu $a[a[2]]$, ale z milczącego założenia, że jeżeli takie wyrażenie pojawi się po lewej stronie przypisania, to powinno być potraktowane jako zmienna. Zresztą od zarania informatycznych dziejów mówi się wprost o „zmiennych wskaźnikowych” (ang. subscripted variables) w Algolu 60 [61] lub o zmiennych indeksowanych (ang. indexed variables) w Pascalu [48].

Problem de Bakker'a z logiką Hoare'a leży więc w niedoskonałym wyobrażeniu o semantyce języka programowania ze zmiennymi tablicowymi⁹⁷. W **Lingua-2** paradoks de Bakker'a-Hoare'a oczywiście nie wystąpi, gdyż instrukcja

$$a.(a.2) := 1$$

byłaby składniowo niepoprawna. W jej miejsce napiszemy

$$a := \text{change-arr } a \text{ at } a.2 \text{ by } 1 \text{ ee}$$

lub w składni kolokwialnej

$$a := \text{change-arr } a \text{ by } a.2 \leq 1 \text{ ee}$$

Na gruncie semantyki **Lingua-2** łatwo teraz udowodnić poprawność poniższego metaprogramu, co właśnie zostało pokazane kilka linijek wyżej:

```
def let a be arr-type number ee
pre a.1 = 2 and a.2 = 2
    a := put-ele a.2 <= 1 into a ee
post a.1 = 2 and a.2 = 1
```

Ten program można też formalnie wywieść z reguły 8.5.2-1 w rozdziale 8.5.2. W tym celu należy skorzystać z łatwej do udowodnienia silnej równoważności ogólnej:

⁹⁷ W modelu denotacyjnym opisanym przez M. Gordona [45] dopuszcza się zmienne indeksowane, jednakże kosztem dość poważnej komplikacji modelu, przez wprowadzenie rozróżnienia na „wyrażenia lewostronne”, które wyliczają adresy i „wyrażenia prawostronne”, które wyliczają wartości. U Gordona w stanach identyfikatorom są przypisywane adresy, a adresom wartości.

```
ide := wyd @ (wrn-1 and wrn-2)
```

```
≡
```

```
(ide := wyd @ wrn-1) and (ide := wyd @ wrn-2)
```

Teraz reguła 8.5.2-1 gwarantuje poprawność metaprogramu

```
def let a be arr-type number ee
pre
  a := change-arr a by a.2 <= 1 ee @ a.1 = 2
and
  a := change-arr a by a.2 <= 1 ee @ a.2 = 1
  a := change-arr a by a.2 <= 1 ee
post a.1 = 2 and a.2 = 1
```

Aby przekształcić ten metaprogram do oczekiwanej postaci, należy skorzystać z dwóch silnych równoważności zachodzących przy założeniu (realizowanym przez preambułę), że *a* jest tablicą liczb:

```
a := change-arr a by a.2 <= 1 ee @ a.1 = 2
≡
a.2 ≠ 1 and a.1 = 2
```

oraz

```
a := change-arr a by a.2 <= 1 ee @ a.2 = 1
≡
a.2 = 2
```

Aby doprowadzić do oczekiwanego w naszym programie prewarunku korzystamy teraz z prawdziwości metaimplikacji:

```
a.1 = 2 and a.2 = 2
⇒
a.2 ≠ 1 and a.1 = 2 and a.2 = 2 and
```

oraz reguły 8.5.2-5 z rozdziału 8.5.2, która pozwala zastąpić każdy prewarunek, warunkiem od niego silniejszym.

8.5 Konstruowanie metaprogramów poprawnych

8.5.1 Konwencja notacyjna

Jak już było mówione, programowanie walidujące polega na budowaniu metaprogramów całkowicie poprawnych z ich całkowicie poprawnych składowych. Na poziomie algebry relacji ta idea została opisana w rozdziale 3.6. W przypadku metaprogramów polega ona na posługiwaniu się regułami konstrukcyjnymi postaci:

```

założenie-1
...
założenie-n
-----
wniosek-1
...
wniosek-m

```

które rozumiemy w ten sposób, że jeżeli są spełnione wszystkie założenia, to są spełnione wszystkie wnioski. Jeżeli wynikanie zachodzi w obie strony, to strzałka ma groty z obu stron.

Czym zatem są założenia i wnioski? Na pierwszy rzut oka mogłoby się wydawać, że są to tezy opisane w rozdziale 8.4. Jednakże tezy dotyczą zawsze konkretnych warunków i programów, a nam są potrzebne reguły ogólne budowane na wzór reguł z rozdziału 3.6. W tamtych regułach występują związane kwantyfikatorami zmienne przebiegające relacje i zbiory. Założeniami i wnioskami w regułach są więc formuły dwuwartościowej logiki klasycznej, które jako argumenty przyjmują relacje i zbiory. W przypadku metaprogramów założenia i wnioski będą podobnymi formułami, tyle że przyjmującymi jako argumenty obiekty składniowe występujące w metaprogramach.

8.5.2 Reguły podstawowe

Przystępując do tworzenia reguł budowania metaprogramów poprawnych (skr. MPP) warto przypomnieć następujące fakty:

- (1) każda (osiągalna) instrukcja jest konserwatywna (rozdział 6.1.4), co oznacza, że jest transparentna względem błędów oraz zmieniając wartości przypisane identyfikatorom w wartościowaniu zachowuje koherentność ich korpusów,
- (2) każdy konstruktor instrukcji jest rzetelny, co oznacza, że zachowuje konserwatywność swoich argumentów.
- (3) jeżeli w danym stanie jakikolwiek warunek jest spełniony, to ten stan nie niesie komunikatu błędu,
- (4) instrukcje nie wyprowadzają poza zbiór stanów adekwatnych dla preambuły.

W poniższych regułach milcząco zakładamy, że wszystkie metazmienne oznaczające preambuły, warunki i instrukcję są związane kwantyfikatorami ogólnymi stojącymi przed diagramem reguły, tj. przed występującą w niej metaimplikacją.

Reguła 8.5.2-1 Przypisanie

```

is-correct (pab)
-----
def pab
pre (ide:=wyd)@ wrn
  ide:=wyd
post wrn

```


Jeżeli preambuła `pab` jest poprawna oraz jest spełniony warunek algorytmiczny $(ide := wyd) @ wrn$, to wprost z definicji $@$ wynika, że przypisanie się wykona, a stan wynikowy spełni warunek `wrn`.

Zauważmy, że poprawność preambuły jest warunkiem koniecznym i dostatecznym poprawności stojącego pod kreską metaprogramu, co prowadzi do z pozoru paradoksalnego wniosku, że bez względu na treść preambuły każdy stojący pod kreską metaprogram będzie poprawny. Ten wniosek wynika jednak stąd, że spełnienie prewarunku implikuje dwa fakty:

- wykonanie `wyd` się zakończy i nie wygeneruje błędu,
- przypisanie nie wygeneruje błędu, a więc `ide` jest zadeklarowany i typu zgodnego z `wyd`.

Na mocy Lematu 8.4.2-9 tych dodatkowych warunków nie musimy do prewarunku dopisywać.

A oto przykład wygenerowania programu na podstawie tej reguły przy odwołaniu się do silnej równoważności:

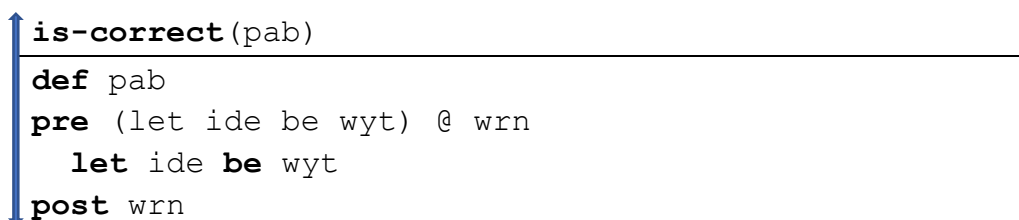
$$x := y + 1 \ @ \ 2 * x > 10 \equiv 2 * (y + 1) > 10 \ \text{and} \ y + 1 \leq \text{max-num}$$

która ma miejsce pod warunkiem, że `max-num` oznacza największą reprezentowalną liczbę. Z tej równoważności możemy wnosić o poprawności następującego programu:

```
def let x, y be number
pre 2*(y+1) > 10
  x:= y+1
post 2*x > 10
```

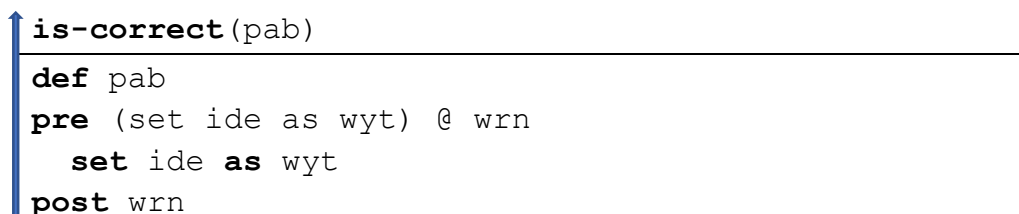
W analogiczny sposób tworzymy regułę dla deklaracji zmiennej i definicji typu

Reguła 8.5.2-2 Deklaracja zmiennej



```
is-correct (pab)
def pab
pre (let ide be wyt) @ wrn
  let ide be wyt
post wrn
```

Reguła 8.5.2-3 Definicja typu



```
is-correct (pab)
def pab
pre (set ide as wyt) @ wrn
  set ide as wyt
post wrn
```

Reguła 8.5.2-4 Sekwencja instrukcji

```

def pab pre prw-1 sin-1 post pow-1
def pab pre prw-2 sin-2 post pow-2
pow-1  $\Rightarrow$  prw-2


---


def pab pre prw-1 sin-1; sin-2 post pow-2
def pab pre prw-1 sin-1; asr pow-1 rsa ; sin-2 post pow-2
def pab pre prw-1 sin-1; asr pow-2 rsa ; sin-2 post pow-2

```

Ta reguła pozwala na zbudowanie z dwóch metaprogramów o wspólnej preambule zawierających instrukcje specyfikowane odpowiednio sin-1 i sin-2 trzech różnych wersji metaprogramu zawierającego instrukcję złożoną sin-1; sin-2 . W tym przypadku nasza reguła wynika wprost z Reguły 3.6.1-1. Należy jednak zauważyć, że obecnie wynikanie zachodzi jedynie z góry do dołu, gdyż opuściliśmy egzystencjalne kwantyfikowanie warunków pow-1 oraz prw-2 .

Zauważmy też, że w tym przypadku nie musimy wprowadzać nad kreskę założenia o poprawności pab , gdyż jest ono zawarte implícite w założeniach o poprawności stojących tamże metaprogramów.

Reguła 8.5.2-5 Instrukcja rozgałęzienia if-then-else

```

prw  $\Rightarrow$  wyd or (not wyd)
def pab pre (prw and wyd) sin-1 post pow
def pab pre (prw and not wyd) sin-2 post pow


---


def pab pre prw if wyd then sin-1 else sin-2 fi post pow

```

Ta reguła wynika z Reguły 3.6.1-2. W tym przypadku metawynikanie jest obustronne, gdyż nie musimy, jak w regule 8.5.2-4, konstruować żadnego pośredniego warunku. Zwróćmy uwagę, że metawarunek

$\text{prw} \Rightarrow \text{wyd or (not wyd)}$

oznacza, że ilekroć prewarunek prw jest spełniony, tylekroć wartość wyrażenia danologicznego wyd jest pp lub ff, a więc jest określona, nie jest komunikatem błędu i jest boolowska. Zauważmy też, że w logice dwuwartościowej to wynikanie byłoby tautologią.

Reguła 8.5.2-6 Pętla while-do-od

```

prw  $\Rightarrow$  nzm
nzm  $\Rightarrow$  (wyd or (not wyd))
nzm and (not wyd)  $\Rightarrow$  pow
def pab pre nzm and wyd sin post nzm
def pab pre nzm while wyd do sin od post TT


---


def pab pre prw while wyd do sin od post pow

```

Ta reguła wynika z Reguły 3.6.2-5. Warunek, oznaczony jako nzm to niezmiennik ciała pętli sin . Ostatnie z założeń nad kreską wyraża fakt, że spełnienie niezmiennika w prewarunku gwarantuje zatrzymanie się programu. Oczywiście przy dowodzeniu własności stopu programu możemy skorzystać z Lematu 3.6.2-1.

Z reguł 3.6.1-3, 3.6.1-4 i 3.6.1-5 wynikają wprost następujące reguły:

Reguła 8.5.2-7 Wzmocnienie prewarunku

```
def pab pre prw sin post pow
prw-1  $\Rightarrow$  prw
-----
def pab pre prw-1 sin post pow
```

Reguła 8.5.2-8 Osłabienie postwarunku

```
def pab pre prw sin post pow
pow  $\Rightarrow$  pow-1
-----
def pab pre prw sin post pow-1
```

Reguła 8.5.2-9 Koniunkcja warunków

```
def pab pre prw-1 sin post pow-1
def pab pre prw-2 sin post pow-2
-----
def pab pre (prw-1 and prw-2) sin post (pow-1 and pow-2)
```

Reguła 8.5.2-10 Rozbudowanie preambuły

```
def pab-1 pre prw sin post pow
pab1 separated-from pab2
-----
def pab-1 ; pab-2 pre prw sin post pow
```

8.5.3 Wywołanie procedury imperatywnej

Rozważmy deklarację procedury

```
proc procedure (val pfo-w ref pfo-r)
  pab-pro ; sin
end proc
```

i niech teza o poprawności wywołania tej procedury będzie metaprogramem postaci:

```
def pab-pr                                     (preambuła programu)
pre prw-wy                                     (prewarunek wywołania)
  call procedure (val pak-w ref pak-r)         (8.5.3-1)
```

post pow-wy (postwarunek wywołania)

gdzie pab-pr będzie preambułą programu, w których procedura jest wywoływana. Niech wreszcie

def pab-ci	(preambuła treści)
pre prw-ci	(prewarunek treści)
sin	(8.5.3-2)
post pow-ci	(postwarunek treści)

będzie teżą wyrażającą poprawność treści procedury.

Reguła wyprowadzenia tezy (8.5.3-1) stanowiącej o poprawności wywołania naszej procedury musi się opierać na pięciu założeniach:

Po pierwsze, deklaracja procedury `procedure` — oznaczmy ją przez `dec` — musi występować składniowo w preambule pab-pr programu, w którym procedura będzie wywoływana, tj. musi być spełniona właściwość (rozdział 8.4.1):

`dec is-in pab-pr`

Po drugie, teza (8.5.3-2) opisująca własności treści procedury musi być spełniona.

Po trzecie, w każdym stanie adekwatnym (rozdział 8.2.2) dla preambuły programu pab-pr oraz spełniającym prewarunek wywołania prw-wy, krotka parametrów formalnych i aktualnych musi być dynamicznie zgodna (rozdział 7.2.2), co oznacza, że musi być spełniona metaimplikacja:

ade-for (pab-pr) **and** prw-wy $\Rightarrow$

conformant (pfo-w, pfo-r, pak-w, pak-r)

Ten warunek gwarantuje, że wykonanie przekazania parametrów aktualnych do formalnych wykona się prawidłowo.

Po czwarte, spełnianie prewarunku prw-wy musi gwarantować, że po przekazaniu parametrów do procedury stan początkowy wykonania procedury będzie spełniał prewarunek jej treści prw-ci. Musi zatem zachodzić metaimplikacja:

$\text{prw-wy} \Rightarrow \text{prw-ci}[\text{i-pfo-w/pak-d}, \text{i-pfo-r/pak-w}]$

gdzie $\text{prw-ci}[\text{i-pfo-w/pak-d}, \text{i-pfo-r/pak-w}]$ oznacza warunek prw-ci, w którym identyfikatory parametrów formalnych zastąpiono parametrami aktualnymi⁹⁸.

Po piąte, spełnienie postwarunku treści procedury pow-ci w stanie adekwatnym dla preambuły treści procedury pab-ci musi gwarantować, że wszystkie parametry formalne referencyjne będą miały określone wartości — co jest niezbędne, aby przekazanie parametrów odbyło się bez wygenerowania błędu — oraz że w stanie zewnętrznym będzie spełniony postwarunek wywołania procedury pow-wy. To oznacza zachodzenie metaimplikacji:

⁹⁸ Formalne zdefiniowanie tej transformacji składniowej warunków wymaga dość żmudnej definicji przez indukcję względem gramatyki warunków, którą na tym etapie pomijam. Zwracam uwagę, że gdyby parametry aktualne mogły być dowolnymi wyrażeniami, ta definicja byłaby jeszcze bardziej złożona.

```
ade-for (pab-ci) and pow-ci ⇔
    defined (pfo-r) and pow-wy [pak-r/pak-w]
```

To rozumowanie pozwala na sformułowanie poniższej reguły, która wymaga pewnego komentarza. Otóż pozornie nie ma w niej odniesienia do faktu, że wywoływana procedura może być bezpośrednio, czy też pośrednio, rekurencyjna. I rzeczywiście, formalnie rzecz biorąc, takiego odniesienia nie ma, jednak stojące nad kreską założenie (2) będzie wymagało zmierzenia się z problemem rekurencji, o ile w deklaracji procedury wystąpią jej rekurencyjne wywołania. Tym problemem zajmę się w rozdziale 8.5.4

Reguła 8.5.3-1 Wywołanie procedury imperatywnej

```
(1) dec is-in pab
(2) def pab-ci pre prw-ci sin post pow-ci
(3) ade-for (pab-pr) and prw-wy ⇔
    conformant (pfo-w, pfo-r, pak-w, pak-r)
(4) prw-wy ⇔ prw-ci [i-pfo-w/pak-d, i-pfo-r/pak-w]
(5) ade-for (pab-ci) and pow-ci ⇔
    defined (pfo-r) and pow-wy [pak-r/pak-w]
```

```
def pab-pr
pre prw-wy
    call procedure (val pak-w ref pak-r)
post pow-wy
```

8.5.4 Przypadek procedury rekurencyjnej

Dla każdorazowego skorzystania z opisanej w rozdziale 8.5.3 reguły dotyczącej wywołania procedury trzeba umieć wyprowadzić poprawny metaprogram określony w pkt. 2 reguły:

```
def pab-ci
pre prw-ci
    sin
post pow-ci
```

(8.5.4-1)

Z rekurencją mamy do czynienia wtedy, gdy wewnątrz instrukcji `sin` pojawia się bezpośrednie lub pośrednie wywołanie procedury `procedure`. Wyprowadzenie tezy (8.5.4-1) będzie oczywiście zależało od tego, w jakim kontekście pojawi się w `sin` wywołanie procedury. Poniżej zajmę się przypadkiem prostej rekurencji strukturalnej rozważanym w rozdziale 3.6.2 w wersji opisanej w Regule 3.6.2-4. Odpowiada ona sytuacji, gdy rozważana procedura rekurencyjna P jest najmniejszym rozwiązaniem równania:

$$X = H \ X \ T \mid E.$$

Oczywiście, aby P była funkcją, H , T i E muszą być funkcjami oraz H i E muszą mieć rozłączne dziedziny. Zapiszmy więc nasze równanie w postaci

$$X = [C] \ H \ X \ T \mid [\neg C] \ E.$$

co odpowiada deklaracji postaci

```
proc procedure (val pfo-w ref pfo-r)
  def pab;
  specins
end proc
```

gdzie specins oznacza instrukcję specyfikowaną postaci

```
if wyd
  then
    head; call procedure(pak-w, pak-r) ; tail      (8.5.4-1)
  else
    exit
  fi
```

Reguła 3.6.2-4 dla tego przypadku przyjmie postać:

Reguła 8.5.4-1 Instrukcja z wywołaniem procedury rekurencyjnej

	<pre>(1) ($\forall$sin) def pab pre (prw and wyd) sin post pow implikuje def pab pre (prw and wyd) head ; sin ; tail post pow (2) def pab pre (prw and (not wyd)) exit post pow</pre> <pre>def pab pre prw specins post pow</pre>
---	--

Z tej reguły należy skorzystać przy wyprowadzaniu tezy (2) w Regule 8.5.3-1 przy założeniu, że specins jest specinstrukcją postaci (8.5.4-1).

8.5.5 Wywołanie procedury funkcyjnej

Dla przeprowadzenia dowodu poprawności procedury funkcyjnej można się posłużyć regułą dla procedury imperatywnej (rozdziału 8.5.3). Przypuśćmy, że mamy do czynienia z deklaracją postaci

```
fun ide-n (pfo) prg return wyr-r endproc
```

Aby udowodnić, że wartość wyrażenia wyr-r ma określoną własność

```
property.(wyr-r)
```

w stanie wynikowym wywołania tej procedury, należy udowodnić prawdziwość metaprogramu postaci:

```
def pab
  pre prw
  prg
post property.(ide-r)
```

8.6 Programowanie transformacyjne

8.6.1 Pierwszy przykład

W poprzednim rozdziale zajmowaliśmy się regułami konstruowania metaprogramów poprawnych z poprawnych składowych. Gdyby użyć języka przemysłu motoryzacyjnego, można by powiedzieć, że były to reguły tworzone na potrzeby montowni programów. W niniejszym rozdziale zajmiemy się regułami transformowania metaprogramów dla wzbogacania ich funkcjonalności. Rozpocznę od przykładów, w których pokażę zastosowania przedstawionych wcześniej reguł, ale też posłużę się nowymi regułami, których zasadność zostanie omówiona w kolejnym rozdziale.

Na początek rozważmy przykład pary metaprogramów poprawnych. Niech n i m oznaczają dwie konkretne liczby całkowite dodatnie czyli wyrażenia liczbowe o wartościach stałych⁹⁹. Niech $\text{isr}(n)$ oznacza część całkowitą pierwiastka kwadratowego z n (ang. integer square root), a $\text{iqt}(n, m)$ oznacza część całkowitą ilorazu n przez m (ang. integer quotient).

```
let x be number
pre true
  x := 0;
  while (x+1)2 ≤ n
    do
      x := x+1
    od
post x = isr(n)
```

```
let x be number
pre true
  x := 0;
  while (x+1)*m ≤ n
    do
      x := x+1
    od
post x = iqt(n, m)
```

Każdy z tych metaprogramów przeszukuje zbiór liczb całkowitych dodatnich, liczba za liczbą, w poszukiwaniu oczekiwanego wyniku. Odwołując się ponownie do języka przemysłu motoryzacyjnego można by powiedzieć, że oba są napędzane tym samym silnikiem, którym jest metaprogram:

```
P1: let x be number
pre true
  x := 0;
  while x+1 ≤ k
    do
      x := x+1
    od
post x = k
```

Na tym silniku raz montujemy urządzenie służące do obliczania funkcji isr , a drugi raz do obliczania iqt . W rzeczywistości to „montowanie” jest zastosowaniem pewnej transformacji zmieniającej funkcjonalność metaprogramu, ale zachowującej jego poprawność. A oto jak ona wygląda w przypadku obliczania pierwiastka.

⁹⁹ W związku z tym nie muszą być w programie ani deklarowane, ani inicjalizowane, mogą natomiast występować w wyrażeniach.

Jeżeli $\text{isr}(n)$ oznacza stałą liczbową (wyrażenie) o wartości równej części całkowitej pierwiastka kwadratowego z n , to bezpośrednio z poprawności P1 możemy w sposób oczywisty wnioskować o poprawności programu P2.

```
P2: let x be number
    pre true
      x := 0;
      begin-asr x ≥ 0
        while x+1 ≤ isr(n)
          do
            x := x+1
          od
        end-asr
    post x = isr(n)
```

W tym i w kolejnych metaprogramach części zmodyfikowane w danym kroku zaznaczam kolorem.

Należy podkreślić, że dopisanie obszaru obowiązywania warunku nie jest wynikiem zastosowania którejś z naszych reguł ogólnych. W tym przypadku, podobnie jak stwierdzając, że P1 jest poprawny, musimy to „ręcznie” udowodnić, co jednak w obu tych przypadkach jest bardzo łatwe.

Nasz metaprogram wygląda dość bezsensownie, bo odwołujemy się w nim do liczby $\text{isr}(n)$, którą właśnie mamy wyliczyć. Teraz jednak wyeliminujemy ją z części programistycznej korzystając z równoważności:

$$x+1 \leq \text{isr}(n) \equiv (x+1)^2 \leq n \quad \text{whenever } x \geq 0$$

oraz z Lematu 8.4.3-3, który pozwala na wymianę wyrażenia boolowskiego w **while** na jemu silnie równoważne. W naszym przypadku jest to silna równoważność jedyne w kontekście warunku $x \geq 0$, jednak jego spełnienie gwarantuje jego dekretacja w programie.

Po przeprowadzeniu opisanych wyżej transformacji otrzymujemy końcowy metaprogram P3, z którego usunęliśmy już dekretacje warunków:

```
P3: let x be number
    pre true
      x := 0;
      while (x+1)2 ≤ n
        do
          x := x+1
        od
    post x = isr(n)
```

Prześledźmy teraz podobną transformacyjną drogę do programu liczącego $\text{isr}(n)$, ale opartego na znacznie szybszym silniku. Niech $\text{po2}.k$ (k is a power of 2) oznacza predykat wyrażający fakt, że k jest potęgą dwójki, tzn.

$$\text{po2}.k \equiv (\exists m \geq 0) \quad k = 2^m$$

i niech $\text{mag}.k$ (the magnitude of k) oznacza funkcję o wartościach będących potęgami dwójki określoną warunkiem

$$\text{mag}.k \leq k < 2 \cdot \text{mag}.k$$

Innymi słowy, jeżeli $\text{mag}.k = 2^m$ to

$$2^m \leq k < 2^{m+1}$$

Oczywiście, dla każdej k liczba m jest wyznaczona jednoznacznie.

Jak dalej wiadomo, dla każdego $n \geq 0$ istnieje jednoznacznie wyznaczony ciąg zer i jedynek $a_0, \dots, a_p$ stanowiący reprezentację binarną liczby n .

Wychodząc z tych oznaczeń można łatwo udowodnić poprawność następujących dwóch programów:

```
Q1: let z be number
    pre true
      z := 1
    begin-asr po2.z
      while z ≤ mag.k do x:=2*z od
    end-asr
  post z = 2*mag.k
```

```
Q2: let z, x be number
    pre z = 2*mag.k
      x := 0
    while z > 1
      do
        z := z/2;
        if x+z < k then x:=x+z else x:=x fi
      od
  post x = k and z = 1
```

Pierwszy program wylicza kolejne potęgi dwójki aż do osiągnięcia $2 \cdot \text{mag}.k$, a drugi z $2 \cdot \text{mag}.k$ wraca do k idąc po kolejnych potęgach 2^m i w trakcie tej drogi dodając do siebie te potęgi, przy których w rozwinięciu binarnym k stoi 1, a więc dla których kolejna potęga „jeszcze się mieści”, tj. $x+z < k$. Zauważmy teraz, że jest prawdziwa następująca równoważność

$$z \leq \text{mag}.k \equiv z \leq k \text{ whenever po2}.z$$

W związku z tym możemy wyrażenie w **while** pierwszego programu zastąpić prawą stroną powyższej równoważności. Teraz uzupełniając preambułę pierwszego programu o deklarację nowej zmiennej x i łącząc oba programy na podstawie reguły 8.5.2-3 otrzymujemy wynikowy program „odszukiwania” k w czasie logarytmicznym. Zauważmy, że poprzedni silnik robił to w czasie liniowym. Możemy też usunąć niepotrzebne już dekrety warunków i przenieść na początek programu inicjalizację zmiennej x .

```
Q3: let z, x be number
    pre true
```

```

z := 1
x := 0
while z ≤ k do z:=2*z od
while z > 1
do
  z := z/2;
  if x+z < k then x:=x+z else x:=x fi
od
post x = k and z = 1

```

Jeżeli w tym programie wyrażenie o wartości stałej k zastąpimy wyrażeniem o wartości stałej $\text{isr}(n)$ to otrzymujemy program liczący $\text{isr}(n)$, ale odwołujący się do tej liczby. Teraz jednak możemy postąpić analogicznie jak w poprzednim przykładzie by wyeliminować $\text{isr}(n)$ z wyrażeń boolowskich w pierwszym **while** oraz w **if**. W tym celu korzystamy z dwóch silnych równoważności warunkowych:

$$\begin{aligned}
 z \leq \text{isr}(n) &\equiv z^2 \leq n && \text{whenever } z > 0 \\
 x+z < \text{isr}(n) &\equiv (x+z)^2 < n && \text{whenever } (z > 0 \text{ and } x \geq 0)
 \end{aligned}$$

Dopisujemy też oczywistą dekrretację warunku pozwalającego na zastosowanie obu transformacji i otrzymamy program liczący $\text{isr}(n)$ w czasie logarytmicznym

Q4: **let** z, x **be** number
pre true
 $z := 1$
 $x := 0$
begin-asr $z > 0 \text{ and } x \geq 0$
 while $z^2 \leq n$ do $z:=2*z$ od
 $x := 0$
 while $z > 1$
 do
 $z := z/2;$
 if $(x+z)^2 < n$ then $x:=x+z$ else $x:=x$ fi
 od
end-asr
post $x = \text{isr}(n)$ and $z = 1$

Ten program poddamy teraz optymalizacjom ograniczającym liczbę zmiennych i liczbę działań. Zaczniemy od obserwacji, że w każdym obiegu pierwszej z pętli program wylicza wartość wyrażenia z^2 , co nie jest optymalne¹⁰⁰. Wprowadzimy więc nową zmienną q i tak uzupełnimy program, aby był spełniony warunek $q = z^2$. Otrzymujemy:

Q5: **let** z, x, q **be** number
pre true

¹⁰⁰ W naszym przypadku ta optymalizacja być może nie jest istotna, ale gdyby q przechowywało aktualizowaną wartość jakiegoś „obliczeniowo kosztownego” wyrażenia, to oszczędność czasu i pamięci mogłaby być nie do pogardzenia.

```

z := 1;
x := 0;
q := 1;
begin-asr z > 0 and x ≥ 0 and q = z2
  while q ≤ n
    do
      off z:=2*z; q:=4*q on
    od
  while z > 1
    do
      off z := z/2; q := q/4 on
      if x2+2*x*z+q ≤ n then x:=x+z else x:=x fi
    od
  end-asr
post x = isr(n) and z = 1 and q = z2

```

Zwróćmy uwagę na dwukrotne użycie wyłączenia **off-on**. Było to konieczne, bo za każdym razem pierwsze przypisanie psuje spełnialność warunku $q=z^2$, a drugie go naprawia. Teraz dokonujemy kolejnych przekształceń:

1. korzystamy z równoważności $z>1 \equiv q>1$ **whenever** ($z>0$ and $q=z^2$) by zmodyfikować wyrażenie boolowskie w drugiej pętli,
2. wprowadzamy dwie nowe zmienne y i p z warunkami $y = n-x^2$ oraz $p = x*z$,
3. korzystamy z równoważności

$$x^2 + 2*x*z + q \leq n \equiv 2*p+q \leq y \text{ whenever } (y=n-x^2 \text{ and } p=x*z)$$

Q6: let z, x, q, y, p be number

```

pre true
z := 1;
x := 0;
q := 1;
begin-asr z > 0 and x ≥ 0 and q = z2
  while q ≤ n
    do
      off z:=2*z; q:=4*q on
    od
  y := n;
  p := 0;
  begin-asr y = n-x2 and p = x*z
    while q > 1
      do
        off z:=z/2; q:=q/4; p:=p/2; on
        if 2*p+q ≤ y
          then x:=x+z; p:=p+q; y:=y-2p-q
          else x:=x
        fi
      od
    end-asr

```

```

end-asr
post x=isr(n) and z=1 and q=z2 and y=n-x2 and p=x*z

```

O ile wcześniejsze wprowadzanie nowych zmiennych i związanych z nimi warunków było — co do intencji — dość oczywiste, a tyle tym razem czytelnik może zadawać sobie pytanie o zamierzenie, dla którego wprowadzono „p” i „y”. Odpowiedź na to pytanie kryje się w dobrze znanej prawdzie, że pisanie programów — podobnie jak gra w szachy — wymaga niekiedy sztuki przewidywania przyszłych ruchów. A jakie to będą ruchy, okaże się nieco dalej.

W kolejnym przekształceniu przygotujemy program do usunięcia z niego zmiennej „z”. W tym celu wykonany następujące przekształcenia:

1. korzystamy z równoważności $q=z^2 \Leftrightarrow \text{isr}(q)=z$ **whenever** $z>0$ by dokonać odpowiedniej zamiany w warunku,
2. korzystamy z warunku $\text{isr}(q)=z$ by zastąpić z przez $\text{isr}(q)$ wszędzie poza lewą stroną znaku przypisania,
3. dokonujemy oczywistych zmian w postwarunku korzystając z faktu, że $z=1$.

```

Q7: let z, x, q, y, p be number
pre true
  z := 1;
  x := 0;
  q := 1;
begin-asr z > 0 and x ≥ 0 and isr(q)=z
  while q ≤ n
  do
    off z:=2*isr(q); q:=4*q on
  od

  y := n;
  p := 0;
begin-asr y = n-x2 and p = x*isr(q)
  while q > 1
  do
    off z:=(isr(q))/2; q:=q/4; p:=p/2 on
    if 2*p+q ≤ y
    then x:=x+isr(q) ; p:=p+q; y:=y-2p-q
    else x:=x
    fi
  od
end-asr
end-asr
post x=isr(n) and z=1 and q=1 and y=n-x2 and p=x

```

Zauważmy teraz, że w Q7 zmienna „z” nie występuje ani w wyrażeniach boolowskich ani po prawych stronach przypisań niezmiennających „z”. Ponieważ nie interesuje nas końcowa wartość „z”, tę zmienną możemy z programu usunąć wraz ze zmieniającym ją przypisaniem (ogólna zasada w rozdziale 8.6.2). W ten sposób otrzymujemy:

Q8: **let** x, q, y, p **be** number
pre true
 $q := 1;$
 $x := 0;$
begin-asr $x \geq 0$
 while $q \leq n$
 do
 $q := 4 * q$
 od
 $y := n;$
 $p := 0;$
begin-asr $y = n - x^2$ **and** $p = x * \text{isr}(q)$
 while $q > 1$
 do
 off $q := q/4; p := p/2$ **on**
 if $2 * p + q \leq y$
 then $p := p + q; y := y - 2p - q$
 else $x := x$ **fi**
 od
 end-asr
end-asr
post $x = \text{isr}(n)$ **and** $q = 1$ **and** $y = n - x^2$ **and** $p = x$

Teraz skorzystamy z równoważności

$$x = \text{isr}(n) \text{ and } p = x \equiv p = \text{isr}(n) \text{ and } p = x$$

by odpowiednio zmodyfikować postwarunek, co spowoduje, że zmienna „ x ” stanie się zbędna, podobnie jak „ z ” w Q7. Możemy ją zatem usunąć otrzymując:

Q9: **let** k, q, y, p **be** number
pre true
 $q := 1;$
while $q \leq n$
 do
 $q := 4 * q$
 od
 $y := n;$
 $p := 0;$
begin-asr $y = n - p^2/q$
 while $q > 1$
 do
 off $q := q/4; p := p/2$ **on**
 if $2 * p + q \leq y$
 then $p := p + q; y := y - 2p - q$
 else $x := x$ **fi**
 od
 end-asr
post $p = \text{isr}(n)$ **and** $q = 1$ **and** $y = n - p^2/q$

W ostatnim kroku

1. usuniemy z postwarunku redundantny warunek $y = n - p^2/q$,
2. usuniemy dekreację warunku $y = n - p^2/q$,
3. w związku z usunięciem obszaru obowiązywania warunku usuniemy wyłączenie **off-on**,
4. instrukcję

$p := p/2$;

if $2 * p + q \leq y$ **then** $p := p + q$; $y := y - 2p - q$ **else** $x := x$ **fi**

zastąpimy równoważną jej instrukcją

if $p + q \leq y$ **then** $p := p/2 + q$; $y := y - p - q$ **else** $p := p/2$ **fi**

W rezultacie otrzymujemy końcową postać naszego programu:

```
Q10:  let q, y, p be number
      pre true
      q := 1;
      while q ≤ n do q:=4*q od
      y := n;
      p := 0;
      while q > 1
      do
        q:=q/4; p:=p/2
        if p+q≤y
        then p:=p/2+q; y:=y-2p-q
        else p:=p/2
        fi
      od
      post p = isr(n)
```

Ten program napisał w latach 1970 bardzo znany wówczas norweski informatyk Ole-Johan Dahl. Nie wiem, jak do niego doszedł, ale można się spodziewać, że prowadził optymalizację podobną do mojej, tyle że bez sformalizowanych reguł.

Na koniec jedna uwaga pragmatyczna. Otóż informatycy, którzy rozmiary tworzonych przez siebie programów liczą w dziesiątkach i setkach tysięcy linii kodu, mogą odnieść się do przedstawionego przykładu z pewnym lekceważeniem. I rzeczywiście, nie jest to program imponujący wielkością, a także jego optymalizacja nie ma pewnie znaczenia dla większości zastosowań. Jednakże, jeżeli tworzymy mikroprogramy zaszywane w architekturze mikroprocesora i wykonywane setki milionów razy przez setki milionów komputerów, to zarówno ich poprawność¹⁰¹, czasy wykonania, jak i liczba zaangażowanych rejestrów (liczba zmiennych) mogą mieć niebagatelne znaczenie. Przedstawiony przykład pokazuje też pewną ogólną metodę budowania programów, na którą składają się trzy kroki:

¹⁰¹ Firma Intel pewnie do dziś pamięta zdarzenie, które wstrząsnęło światem IT gdzieś chyba na początku lat 1990, gdy odkryto, że ich mikroprocesor zainstalowany w milionach PC popełniał błędy przy mnożeniu.

1. napisanie programu „silnika”, który przeszukuje jakiś obszar danych,
2. nałożenia na ten silnik mechanizmów wykonujących określone przetwarzanie danych,
3. optymalizacja programu.

Jak zobaczymy w rozdziale 8.6.3, optymalizacja programu może polegać też na zmianie typu danych, które podlegają przetwarzaniu.

8.6.2 Dodawanie identyfikatora rejestrowego

W tym rozdziale opiszę w przypadku ogólnym transformację polegającą na takim wzbogaceniu metaprogramu, aby na pewnym jego obszarze był niezmiennie spełniony warunek postaci

$$\text{ide-r} = \text{wyd-r}$$

Tego typu transformacja była kilkakrotnie użyta w rozdziale 8.6.1, np. w przejściu od Q4 do Q5.

Identyfikator *ide* spełniający warunek postaci *ide=wyd* na jakimś obszarze obowiązywania tego warunku nazwiemy *identyfikatorem rejestrowym* lub *rejestrem*, wyrażenie *wyd* — *wyrażeniem rejestrowym*, a warunek *ide=wyd* — *warunkiem rejestrowym*.

Zacznijmy od zupełnie naturalnego rozszerzenia wprowadzonej w rozdziale 8.2.4 operacji @ z warunków na dowolne wyrażenia danologiczne:

$$\text{Swd.}[\text{sin} @ \text{wyd}] = \text{Ssi.}[\text{sin}] \bullet \text{Swd.}[\text{wyd}]$$

Rozważmy teraz metaprogram P postaci

```
P: def pab
  pre prw
    ins-h ;                                (głowa (head) programu; może być pusta)
    asr wrn-p rsa ;                        (warunek początkowy)
    begin-asr wrn-r ;                      (warunek rejestrowy)
      ins
    end-asr ;
    ins-t                                (ogon (tail) programu ; może być pusty)
  post pow
```

i założmy o nim, że jest poprawny. Niech *wyd-r* będzie wyrażeniem danologicznym spełniającym metaimplikacje

$$\text{wrn-p} \Rightarrow \text{defined}(\text{wyd-r}) \quad \text{oraz}$$

$$\text{wrn-r} \Rightarrow \text{defined}(\text{wyd-r})$$

niech *ide-r* będzie identyfikatorem, który nie występuje w P i niech *dez-ide-r* będzie taką deklaracją zmiennej *ide-r*, aby ta zmienna była typologicznie zgodna w wyrażeniu

wyd-r. Transformacja wzbogacania metaprogramu o warunek rejestrowy $ide-r = wyd-r$ polega na przekształceniu P do postaci:

```
Q: def pab ; dez-ide-r
  pre prw
    ins-h ;
    ide-r := wyd-r ;
    begin-asr wrn-r and ide-r = wyd-r
      ins $ (ide-r=wyd-r)           (instrukcja wzbogacona; patrz niżej)
    end-asr ;
    ins-t
  post pow
```

gdzie $ins\ \$\ (ide-r=wyd-r)$ jest taką modyfikacją (rozbudowaniem) instrukcji ins , aby program Q był poprawny, gdy tylko P jest poprawny. Asercja **asr** wrn-p **rsa** nie pojawia się w Q (choć oczywiście można by ją tam pozostawić), gdyż służyła jedynie do wyrażenia faktu, że w danym miejscu programu wartość wyrażenia $wyd-r$ jest określona. Teraz ten fakt wynika ze spełnienia asercji.

Składniową operację $\$$ definiujemy przez indukcję względem struktury ins . Zaczniemy od przypadku, gdy ins składa się z pojedynczego przypisania i jest postaci

$ide := wyd$

gdzie oczywiście $ide \neq ide-r$, gdyż z założenia $ide-r$ nie występuje w P .

Jeżeli ide nie występuje w $wyd-r$, to wykonanie powyższego przypisania nie spowoduje zmiany wartości wyrażenia $wyd-r$, nie musimy więc dopisywać do instrukcji żadnego uaktywnienia.

Jeżeli jednak tak nie jest, to bezpośrednio po przypisaniu $ide := wyd$ musimy dodać przypisanie przywracające spełnienie warunku $ide-r = wyd-r$. Zatem w tym przypadku

$(ide := wyd)\ \$\ (ide-r = wyd-r) =$
off $ide := wyd ; ide-r := wyd-r$ **on**

Wyłączenie warunku zostaje tu użyte, bo skoro ide występuje w $wrn-r$, to zmiana jego wartości może powodować zmianę wartości $wrn-r$, a więc sfalsyfikować ten warunek. W przypadku przekształcania metaprogramu $Q4$ na $Q5$ przy warunku rejestrowym $q = z^2$ prowadzi to do rozbudowania instrukcji $z := 2 * z$ do instrukcji:

off $z := 2 * z ; q := z^2$ **on**

Tę instrukcję można teraz zamienić na równoważną jej

off $q := ((z := 2 * z) @ z^2) ; z := 2 * z$ **on**

z której po wyeliminowaniu znaku operacji $@$, czyli sprowadzeniu wyrażenia $(z := 2 * z) @ z^2$ do postaci standardowej, otrzymujemy:

```
off q:=4*z2 ; z:=2*z on
```

a ponieważ przed instrukcją jest spełniony warunek $q=z^2$, więc na mocy tego faktu możemy zamienić powyższą instrukcję na

```
off z:=2*z ; q:=4*q on
```

W przypadku ogólnym te przekształcenia mają następujący przebieg. Wpierw instrukcję

```
off ide:=wyd ; ide-r:=wyd-r on
```

zamieniamy na równoważną jej

```
off ide-r:=((ide:=wyd) @ wyd-r) ; ide:=wyd on
```

Dalej, wyrażenie $((ide:=wyd) @ wyd-r)$ przekształcamy do postaci standardowej, a następnie, o ile to możliwe, staramy się tak przekształcić nasze wyrażenie, aby dało się z niego wyeliminować identyfikator *ide* na rzecz *ide-r* korzystając z warunku rejestrowego $ide-r=wyd-r$.

Drugi przypadek wyjściowy, którym należy się zająć, to instrukcja wywołania procedury, której ogólna postać jest następująca;

```
call ide(ref pak-r val pak-w)
```

Przyjmijmy, że chcemy wprowadzić do programu dekretację warunku $ide-r=wyd-r$ i założmy, że powyższa instrukcja wywołania procedury występuje w identycznym kontekście jak ten, w którym występowała instrukcja przypisania. Wtedy ponownie mamy dwa przypadki do rozważenia. Jeżeli żaden z aktualnych parametrów referencyjnych wywołania nie występuje w *wyd-r*, to pozostawiamy instrukcję niezmienną, a jeżeli tak nie jest zastępujemy ją instrukcją

```
off call ide (ref pak-r val pak-w); ide-r:=wyd-r on.
```

W ten sposób kończymy pierwszy krok indukcji strukturalnej. Kolejne kroki są już całkiem oczywiste:

```
[ide-1 ; ide-2]$[ide-r=wyd-r] =  
ide-1 $[ide-r=wyd-r] ; ide-2 $[ide-r=wyd-r]
```

```
[if wyd-b then ins-1 else ins-2 fi ]$[ide-r=wyd-r] =  
if wyd-b then ins-1 $[ide-r=wyd-r] else ins-2 $[ide-r=wyd-r]  
fi
```

```
[while wyd-b do ins od ] $ [ide-r=wyd-r] =  
while wyd-b do ins $ [ide-r=wyd-r] od
```

Mówiąc wprost, po każdej instrukcji przypisania i instrukcji wywołania procedury, których wykonanie zmienia wartość wyrażenia rejestrowego, dopisujemy instrukcję przypisania przywracającą spełnianie warunku rejestrowego. Rozszerzenie **\$** z instrukcji na specinstrukcje jest już dość oczywiste.

Na koniec zwróćmy uwagę na metodologiczną różnicę pomiędzy dwiema operacjami składniowymi z poziomu meta — @ oraz \$ — które są odpowiednio typów:

@ : Instrukcja x WyrDan $\mapsto$ WyrDan

\$: Instrukcja x WyrDan $\mapsto$ Instrukcja

Pierwsza operacja tworzy wyrażenia, np.

$(z := 2 * z) @ z^2$

będące elementami składni **LinguaV-2.0**, a więc klauzula

Instrukcja @ WyrDan

występuje w gramatyce języka **Lingua-2V**. Symbol @ jest składniowym odpowiednikiem operacji na denotacjach @ tak jak średnik ';' jest składniowym odpowiednikiem sekwencyjnego składania funkcji '•'.

Sytuacja z operatorem \$ jest inna, gdyż nie ma on swojego odpowiednika na poziomie składni, a więc wyrażenie typu

$(z := 2 * z) \$ q = z^2$

nie byłoby na poziomie składni uprawnione. Metawyrażenie

$(z := 2 * z) \$ q = z^2$

trzeba przekształcić do uprawnionego odwołując się do indukcyjnej definicji operatora \$.

8.6.3 Zmiana typu danych

Innym zastosowaniem techniki warunków rejestrowych jest transformacja programu z jednego typu danych na drugi. W niniejszym rozdziale pokażemy jak przekształcić program Q10 z rozdziału 8.6.2 w program operujący na binarnych reprezentacjach liczb. Niech Binarna będzie zbiorem reprezentacji binarnych liczb całkowitych, tj. zerojedynekowych krotek (rozdział 2.1.4).

$\text{bin} : \text{Binarna} = \{(1)\} \odot \{(0), (1)\}^{c*}$

Na tym zbiorze określimy kilka funkcji i relacji:

$\text{sl} : \text{Binarna} \mapsto \text{Binarna}$ (shift left)

$\text{sl.bin} =$

$\text{bin} = (0) \rightarrow 0$

prawda $\rightarrow \text{bin} \odot (0)$

$\text{sr} : \text{Binarna} \mapsto \text{Binarna}$ (shift right)

$\text{sr.bin} =$

$\text{bin} = (0) \rightarrow 0$

prawda $\rightarrow \text{pop.bin}$

$+$: Binarna $\mapsto$ Binarna (dodawanie)

- $- : \text{Binarna} \mapsto \text{Binarna}$ (odejmowanie)
- $< : \text{Binarna} \mapsto \{\text{pp}, \text{ff}\}$ (wcześniejszy)
- $\leq : \text{Binarna} \mapsto \{\text{pp}, \text{ff}\}$ (wcześniejszy lub równy)

Dodawanie i odejmowanie krotek oznaczamy tymi samymi symbolami co odpowiadające im operacje na liczbach i zakładamy, że są zdefiniowane w ten sposób, aby były spełnione równania (5) i (6) poniżej. Symbole porządków odpowiadają porządkowi leksykograficznemu i są ponownie tymi samymi symbolami, których używamy w stosunku do liczb.

- $\text{b2n} : \text{Binarna} \mapsto \text{Liczba}$ (*binary to number*; funkcja konwersji)
- $\text{n2b} : \text{Liczba} \mapsto \text{Binarna}$ (*number to binary*; funkcja konwersji)

Wszystkie powyższe funkcje i relacje są określone w ten sposób, aby były spełnione poniższe zależności:

- (1) $\text{b2n}(\text{n2b}(\text{lic})) = \text{lic}$ gdzie $\text{lic} : \text{Liczba}$
- (2) $\text{n2b}(\text{b2n}(\text{bin})) = \text{bin}$
- (3) $\text{n2b}(\text{lic} * 2) = \text{sl}(\text{n2b}(\text{lic}))$
- (4) $\text{n2b}(\text{lic} / 2) = \text{sr}(\text{n2b}(\text{lic}))$ „/” oznacza część całkowitą z dzielenia
- (5) $\text{n2b}(\text{lic1} + \text{lic2}) = \text{n2b}(\text{lic1}) + \text{n2b}(\text{lic2})$
- (6) $\text{n2b}(\text{lic1} - \text{lic2}) = \text{n2b}(\text{lic1}) - \text{n2b}(\text{lic2})$
- (7) $\text{n2b}(\text{lic1}) < \text{n2b}(\text{lic2}) \quad \text{wtw} \quad \text{lic1} < \text{lic2}$
- (8) $\text{n2b}(\text{lic1}) \leq \text{n2b}(\text{lic2}) \quad \text{wtw} \quad \text{lic1} \leq \text{lic2}$

Tak przygotowani przekształcamy program Q10 przez wprowadzenie do niego trzech nowych zmiennych i trzech odpowiadających im warunków rejestrowych:

- $Q = \text{n2b}(q)$
- $Y = \text{n2b}(y)$
- $P = \text{n2b}(p)$

Wprowadzamy też do języka nowy typ danych `binary`. Nowy program liczy równolegle na liczbach i na ich reprezentacjach binarnych. Wprowadzamy do niego dekrety warunków i przesuwamy do przodu inicjalizację wszystkich zmiennych:

Q11: def

```

    let n, q, y, p be number
    let Q, Y, P be binary
pre n ≥ 1
    q := 1; Q := (1);
    y := n; Y := n2b(n);
    p := 0; P := (0);
    begin-asr Q = n2b(q) and Y = n2b(y) and P = n2b(p)
```

```

while  $q \leq n$  do off  $q := 4 * q$  ;  $Q = sl(sl(Q))$  on od
while  $q > 1$ 
  do
    off  $q := q/4$ ;  $p := p/2$ ;
       $Q := sr(sr(Q))$ ;  $P := sr(P)$  on
    if  $p + q \leq y$ 
      then off  $p := p/2 + q$ ;  $y := y - 2p - q$ ;
         $P := sr(P) + Q$ ;  $Y := Y - sl(P) - Q$  on
      else off  $p := p/2$ ;  $P := sr(P)$  on
    fi
  od
end-asr
post  $p = isr(n)$  and  $q = 1$ 

```

Korzystamy teraz z czterech równoważności warunkowych:

$$\begin{aligned}
 q \leq n & \quad \equiv Q \leq n2b(n) \textbf{ whenever } Q = n2b(q) \\
 q > 1 & \quad \equiv (1) < Q \quad \textbf{ whenever } Q = n2b(q) \\
 p + q \leq y & \quad \equiv P + Q \leq Y \quad \textbf{ whenever } Q = n2b(q) \textbf{ and } Y = n2b(y) \\
 & \quad \quad \quad \textbf{ and } P = n2b(p) \\
 p = isr(n) & \quad \equiv P = n2b(isr(n)) \textbf{ whenever } P = isr(p)
 \end{aligned}$$

by wymienić boolowskie wyrażenia liczbowe na binarne, a następnie usuwamy z programu wszystkie zmienne liczbowe poza n i odpowiadające im przypisania, a także dekretyzację warunku. Ponieważ obszar obowiązywania warunku sięga do końca programu, możemy też odpowiednio zmodyfikować postwarunek.

Q12: **def**

```

  let  $n$  be number
  let  $Q, Y, P$  be binary
pre  $n \geq 1$ 
   $Q := (1)$ ;
   $Y := n2b(n)$ ;
   $P := (0)$ ;
  while  $Q \leq N$  do  $Q = sl(sl(Q))$  od
  while  $(1) < Q$ 
    do
       $Q := sr(sr(Q))$ ;  $P := sr(P)$ 

```

```

if P+Q≤Y
  then P:=sr(P)+Q; Y:=Y-sl(P)-Q
  else P:=sr(P)
fi
od
post P = n2b(isr(n)) and Q = (1)

```

8.7 Niezmienniki versus asercje

Z filozoficznego punktu widzenia niezmienniki i asercje, tak jak zostały zdefiniowane w niniejszej książce, są zbliżone do niezmienników w sensie R. Floyda [40] i C.A.R. Hoare’a [47]. Jednak formalnie są to pojęcia nie tylko między sobą różne, ale też należące do istotnie różnych kategorii językowych¹⁰².

Nasze niezmienniki to warunki (rozdział 8.2), przy czym pojęcie niezmiennika dotyczy w zasadzie relacji pomiędzy warunkiem a instrukcją. Mówimy, że *warunek wrn jest niezmiennikiem instrukcji ins* jeżeli spełnia jedną z dwóch metaimplikacji (rozdział 8.4.2):

```

wrn @ ins ⇔ wrn          — słaby niezmiennik
wrn          ⇔ ins @ wrn  — mocny niezmiennik

```

Słabe niezmienniki występują w dowodach częściowej poprawności programów, mocne — w dowodach poprawności całkowitej.

Nieco inaczej jest rozumiane pojęcie *niezmiennika pętli while*, które pojawia się w regule 8.5.2-6:

```

prw ⇔ nzm
nzm ⇔ (wyd or (not wyd))
nzm and (not wyd) ⇔ pow
def pab pre nzm and wyd      sin post nzm
def pab pre nzm while wyd do sin od post TT
def pab pre prw while wyd do sin od post pow

```

W tym przypadku, aby warunek nzm został uznany za niezmiennik pętli

```
while wyd do sin od,
```

musi spełniać wszystkie metawarunki stojące w regule nad kreską.

W obu więc przypadkach niezmienniki nie należą do warstwy programistycznej metaprogramu, a jedynie do jego części deskryptywnej. W konsekwencji nie mają swoich odpowiedników ani w składni ani w denotacjach specinstrukcji.

Sytuacja z asercjami jest inna. Po pierwsze, nie są one warunkami, ale specinstrukcjami budowanymi z warunków. Specinstrukcja

¹⁰² Na potrzebę omówienia tych różnic zwrócił moją uwagę Stefan Sokołowski.

asr wrn rsa

działa jak filtr, który nie zmienia stanu, gdy stan spełnia warunek `wrn`, i który generuje błąd (czyli wpisuje błąd do stanu) w przeciwnym przypadku.

O ile niezmienniki są wykorzystywane w dowodach poprawności metaprogramów, o tyle asercjami posługujemy się przy transformowaniu metaprogramów poprawnych w metaprogramy poprawne.

Asercje opisują lokalne właściwości programów wyrażane przez właściwości ich lokalnie pojawiających się stanów. Wykorzystanie asercji do transformowania metaprogramów opiera się na obserwacji, że jeżeli dany metaprogram jest poprawny, to zawarte w nim asercje muszą być spełnione w każdym jego przebiegu rozpoczynającym się od stanu spełniającego prewarunek. Ta właśnie teza pozwala nam decydować, czy i które reguły transformacyjne można do danego programu zastosować¹⁰³.

Z asercjami wiążą się dwa pojęcia im pochodne — *obszar obowiązywania warunku* i *obszar wyłączenia obowiązywania* — a także związane z nimi środki językowe *dekretowania* takich obszarów (rozdział 8.3):

begin-asr wrn; sin end-asr

off-asr sin on-asr.

Te konstrukcje zostały zdefiniowane jako kolokwializmy, a więc nie występują ani na poziomie składni konkretnej i denotacji (jak asercje), ani na poziomie meta (jak warunki).

¹⁰³ W przykładach przedstawionych w rozdziale 8.6 asercje były wykorzystane jedynie w transformacjach związanych z wprowadzaniem identyfikatorów rejestrowych. Być może jednak znajdą one zastosowania w innych sytuacjach.

9 Lingua-3 — programowanie obiektowe

9.1 Założenia modelu

W **Lingua-2** użytkownik języka otrzymuje do dyspozycji dwie algebry podstawowe — kompozytów i typów — oraz zbudowaną nad nimi algebrę denotacji. To instrumentarium może wzbogacać definiując własne typy, procedury i funkcje, które umieszcza w środowisku. Jednakże wszystko co zbuduje w ten sposób pozostaje dostępne tylko w obszarze stworzonego przez niego programu.

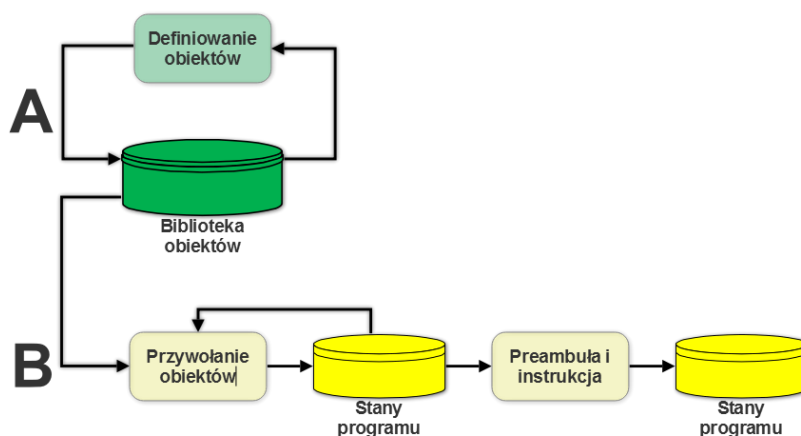
Idea programowania obiektowego służy temu, by instrumentarium stworzone przez jednego użytkownika mogło być udostępniane innym jako nieustannie rozwijany element wyposażenia języka. Tak pomyślane instrumentaria będą nazywał *obiektami*. Intuicyjnie każdy obiekt odpowiada pewnej sekwencji definicji typów oraz deklaracji procedur, a w tym procedur funkcyjnych i multiprocedur. Można więc powiedzieć, że obiekty są bliskie preambułom bez deklaracji zmiennych (por. rozdział 6.1.9).

Denotacyjnie obiekty są transformacjami stanów, które modyfikują środowiska przez wiązanie w nich typów i procedur:

$$\text{obi} : \text{Obiekt} = \text{Stan} \mapsto \text{Stan}$$

Ponieważ obiekty będą zapisywane w pamięci komputera wprowadzam pojęcie *biblioteki obiektów*, która nazwom obiektów (identyfikatorom) przypisuje obiekty:

$$\text{bio} : \text{BibObi} = \text{Identyfikator} \Rightarrow \text{Obiekt}$$



Rys. 9.1-1 Dwa typy zadań programistycznych w Lingua-3

Obiekty i biblioteki obiektów posłużą do zdefiniowania narzędzi programistycznych pozwalających na wykonywanie dwóch typów zadań zilustrowanych na Rys. 9.1-1:

- A. budowanie biblioteki, tj. umieszczanie obiektów w bibliotece i ich usuwanie,

B. przywoływanie obiektów w programach.

Intuicyjnie zadanie A polega na definiowaniu obiektów, a następnie umieszczaniu ich w bibliotece pod wskazanymi nazwami. Obiekty są tworzone przez:

1. stworzenie definicji typu lub deklaracji procedury,
2. pobranie obiektu z biblioteki przez wskazanie jego nazwy,
3. sekwencyjne złożenie obiektów

Umieszczone w bibliotece obiekty pozostają w niej aż do ewentualnego usunięcia.

Zadanie B to budowanie programów z umieszczonymi przed preambułą przywołaniami obiektów. Przywołania modyfikują stan przez wpisanie do środowiska zdefiniowanych w obiekcie typów i procedur. Programy w **Lingua-3** składają się z trzech sekwencyjnie ułożonych segmentów: przywołań obiektów, preambuły i instrukcji. W preambułach i instrukcjach przywołania obiektów nie występują.

W hierarchii pojęciowej modelu denotacyjnego obiekty są charakterem zbliżone do danych: ich dziedzina nie jest nośnikiem algebry denotacji, a one same pojawiają się jako wartości *wyrażeń obiektowych* grających rolę podobną do wyrażeń danologicznych. Z kolei biblioteki obiektów są charakteru zbliżonego do stanów, pojawiają się więc jako argumenty *przywołań obiektów* w programach, a ich dziedzina również nie będzie stanowić nośnika algebry denotacji.

9.2 Wyrażenia obiektowe

Analogicznie do wyrażeń danologicznych, które generują dane, *wyrażenia obiektowe* generują obiekty. Aby obiekty można było generować również przez pobieranie ich z biblioteki, przyjmujemy, że denotacje wyrażeń obiektowych są funkcjami z bibliotek w obiekty:

$$\text{dwo} : \text{DenWyrObi} = \text{BibObi} \mapsto \text{Obiekt} \mid \text{Błąd}$$

Jak zobaczymy niżej, te denotacje tworzymy na trzy sposoby:

1. z denotacji definicji typów i deklaracji procedur,
2. przez wskazanie nazwy obiektu w bibliotece,
3. przez sekwencyjne złożenie wyrażeń obiektowych.

Pierwsza grupa konstruktorów odpowiada wyrażeniom o wartościach stałych, tj. nie zależących od biblioteki, a pierwszy z nich tworzy denotację wyrażenia obiektowego, która jako wynikowy obiekt przyjmuje (niezależnie od biblioteki) wejściową denotację definicji typu *ddt*.

$$\text{twórz-dwo-z-def-typ} : \text{DenDefTyp} \mapsto \text{DenWyrObi}$$

$$\text{twórz-dwo-z-def-typ.ddt.bio} = \text{ddt}$$

W analogiczny sposób są definiowane pozostałe trzy konstruktory związane z deklaracjami procedur:

$$\text{twórz-dwo-dek-prf} : \text{DenDekPrf} \mapsto \text{DenWyrObi}$$

$$\text{twórz-dwo-dek-prf.ddf.bio} = \text{ddf} \quad (\text{twórz dwo z deklaracji procedury funkcyjnej})$$

twórz-dwo-dek-pri : DenDekPri $\mapsto$ DenWyrObi
 twórz-dwo-dek-pri.ddp.bio = ddp (twórz dwo z deklaracji procedury imperatywnej)

twórz-dwo-dek-wpri : DenDekWiePri $\mapsto$ DenWyrObi
 twórz-dwo-dek-wpri.ddw.bio = ddw (twórz dwo z deklaracji wieloprocedury)

Wszystkie cztery konstruktory są funkcjami stałymi na bibliotekach i są charakterem zbliżone do identycznościowych włożeń, które znamy już z rozdziału 6.1.9. W tym przypadku denotacje definicji typów i deklaracji procedur „stają się” obiektami.

Kolejny konstruktor tworzy wyrażenie obiektowe, które nazwiemy *odwołaniem do biblioteki* i które wskazuje obiekt w bibliotece przez podanie jego nazwy. Jeżeli takiego obiektu w bibliotece nie ma, to jest generowany błąd:

weź-obi : Identyfikator $\mapsto$ DenWyrObi
 weź-obi.ide.bio =
 bio.ide = ? $\rightarrow$ 'obiekt-niezdefiniowany'
 prawda $\rightarrow$ bio.ide

Zauważmy, że ta konstrukcja umożliwia odwoływanie się w definicji obiektu do obiektów już zdefiniowanych co w jakimś sensie odpowiada mechanizmowi dziedziczenia.

Ostatnim konstruktorem związanym z wyrażeniami obiektowymi jest złożenie sekwencyjne:

sekwencja-dwo : DenWyrObi x DenWyrObi $\mapsto$ DenWyrObi
 sekwencja-dwo.(dwo-1, dwo-2).bio =
 dwo-i.bio : Błąd $\rightarrow$ dwo-i.bio dla i = 1,2
 prawda $\rightarrow$ dwo-1.bio • dwo-2.bio

Wynikiem złożenia dwóch denotacji wyrażeń jest albo błąd albo denotacja, która składa sekwencyjnie obiekty wygenerowane przez dwo-1 i dwo-2.

Składniowo wyrażeniu obiektowemu będzie więc odpowiadała sekwencja definicji typów, deklaracji procedur i odwołań. Te ostatnie pozwalają budować obiekty jako rozszerzenia obiektów wcześniej zapisanych w bibliotece.

9.3 Deklaracje obiektów

Biblioteki są budowane za pomocą *deklaracji obiektów*, które przypisują obiektom nazwy i umieszczają je w bibliotece. Ich dziedzinę definiuję jako zbiór funkcji transformujących biblioteki i ewentualnie emitujących sygnał błędu:

ddo : DenDekObi = BibObi $\mapsto$ BibObi | Błąd

Pierwszym konstruktorem tej dziedziny jest operacja, która dla zadanego identyfikatora i obiektu umieszcza ten obiekt w bibliotece pod zadaną nazwą:

deklaruj-obi : Identyfikator x DenWyrObi $\mapsto$ DenDekObi

```

deklaruj-obi.(ide, dwo).bio =
  bio.ide = ! → 'identyfikator-zajęty'
  niech
    obi = dwo.bio
  obi : Błąd → obi
  prawda → bio[ide/obi]

```

Drugi konstruktor buduje denotacje służące do usuwania obiektów z biblioteki:

```

usuń-obi : Identyfikator ↦ DenDekObi
usuń-obi.ide.bio =
  bio.ide = ? → 'obiekt-niedefiniowany'
  prawda → bio[ide/?]

```

W przypadku deklaracji obiektów nie wprowadzam konstruktora złożenia przyjmując, że każdy „akt” modyfikacji biblioteki jest aktem jednostkowym. To oczywiście decyzja o charakterze inżynierskim, a nie matematyczna konieczność.

9.4 Przywoływanie obiektów w programach

Aby obiekt wykorzystać w programie należy go przywołać, tj. do środowiska początkowego wykonania programu wpisać jego typy i procedury. Dla wprowadzenia tego mechanizmu do modelu definiuję *przywołania obiektów*, których denotacjami są funkcje przypisujące bibliotekom obiekty:

```

dpo : DenPrzObi = BibObi ↦ Obiekt

```

Ta dziedzina jest bardzo podobna do dziedziny denotacji wyrażeń obiektowych, jednakże nie zawiera błędów a w zamian za to jej część osiągalna będzie zawierać pseudoobiekty, których nie ma w DenWyrObi.

Pierwszy konstruktor tej dziedziny z zadanego identyfikatora *ide* tworzy przywołanie obiektu o nazwie *ide*:

```

przywołaj-obi : Identyfikator ↦ DenPrzObi
przywołaj-obi.ide.bio.sta =
  jest-błąd.sta → sta
  bio.ide = ? → sta ◀ 'obiekt-nieznany'
  prawda → bio.ide.sta

```

Jeżeli poszukiwanego obiektu nie ma w bibliotece, to tworzy się *pseudoobiekt*, który każdy stan nie niosący błędu przekształca na stan niosący komunikat 'obiekt-nieznany'. Zauważmy, że przywołanie obiektu może wprowadzić do stanu nie tylko ten komunikat, ale też i błąd, który pojawi się w stanie (bio.ide).sta i będzie pochodził z poziomu definicji typu lub deklaracji procedury.

Przywołania obiektów możemy też składać sekwencyjnie, a ponieważ przywołania atomowe, tj. generowane powyższym konstruktorem, są transparentne względem błędów, to także będą również przywołania złożone.

sekwencja-przy : $\text{DenPrzObi} \times \text{DenPrzObi} \mapsto \text{DenPrzObi}$

sekwencja-przy.(dpo-1, dpo-2).bio = dpo-1.bio • dpo-2.bio

9.5 Prefiksowanie programów przywołaniami obiektów

Aby w programie posłużyć się typami i procedurami zdefiniowanymi w obiektach, trzeba te obiekty przywołać przed wykonaniem programu. Analogicznie jak w przypadku preambuł, które w programach poprzedzają instrukcje (por. rozdziały 6.1.9 oraz 7.7), założymy teraz, że przywołania obiektów stoją przed preambułą. To ponownie nie matematyczna konieczność, ale zasada upraszczająca reguły konstruowania programów poprawnych. Wprowadzamy więc pojęcie *programu prefiksowanego*. Ich dziedzinę denotacji definiuje w sposób następujący:

dpp : $\text{DenProPre} = \text{BibObi} \times \text{Stan} \rightarrow \text{Stan}$

Podstawowy konstruktor programów prefiksowanych tworzy je z przywołania obiektów i programu bez prefiksu:

prefiksuj-program : $\text{DenPrzObi} \times \text{DenPro} \mapsto \text{DenProPre}$

prefiksuj-program.(dpo, dpr).(bio, sta) =

jest-błąd.sta $\rightarrow$ sta

niech

sta-1 = dpo.bio.sta

jest-błąd.sta-1 $\rightarrow$ sta-1

prawda $\rightarrow$ dpr.sta-1

Tak więc wykonanie programu prefiksowanego rozpoczyna się od wykonania jedyne go w nim przywołania obiektów, które jednak może być sekwencją wielu przywołań atomowych. Następnie, jeżeli nie pojawi się błąd, to stan z zagnieżdżonymi obiektami staje się stanem początkowym programu. Zauważmy, że program nie musi mieć dostępu do biblioteki, bo nie wystąpią w nim przywołania obiektów.

Aby programy prefiksowane obejmowały również programy bez prefiksów wprowadzamy konstruktor włożenia:

bez-prefiksu : $\text{DenPro} \mapsto \text{DenProPre}$

bez-prefiksu.dpr = dpr

9.6 Rozszerzenie algebry składni

Algebra denotacji dla **Lingua-3** powstała z rozbudowania algebry denotacji dla **Lingua-2** o cztery następujące nośniki:

dwo : $\text{DenWyrObi} = \text{BibObi} \mapsto \text{Obiekt} \mid \text{Błąd}$

ddo : $\text{DenDekObi} = \text{BibObi} \mapsto \text{BibObi} \mid \text{Błąd}$

dpo : $\text{DenPrzObi} = \text{BibObi} \mapsto \text{Obiekt}$

dpp : $\text{DenProPre} = \text{BibObi} \times \text{Stan} \rightarrow \text{Stan}$

Gramatykę składni abstrakcyjnej dla **Lingua-3** otrzymujemy z gramatyki składni abstrakcyjnej dla **Lingua-2** przez jej rozbudowanie o poniższe cztery klauzule:

wyo : WyrObi = (wyrażenia obiektowe)

```
twórz-dwo-def-typ ( DefinicjaTyp )      |
twórz-dwo-dek-prf ( DekProFun )        |
twórz-dwo-dek-pri ( DekProImp )        |
twórz-dwo-dek-wpri ( DekWieProImp )    |
weź-obi ( Identyfikator )              |
sekwencja-dwo ( WyrObi , WyrObi )
```

deo : DefObi = (deklaracje obiektów)

```
deklaruj-obi ( Identyfikator , WyrObi )|
usuń-obi ( Identyfikator )
```

pob : Przywołanie = (przywołania)

```
przywołaj-obi ( Identyfikator )|
sekwencja-przy ( Przywołanie, Przywołanie)
```

prx : ProgramPrefiksowany = (programy prefiksowane)

```
bez-prefiksu ( Program )                |
prefiksuj-program ( Przywołanie, Program)
```

Od tej składni przechodzimy do ostatecznej składni konkretnej **Lingua-3**:

wyo : WyrObi = (wyrażenia obiektowe)

```
DefinicjaTyp      |
DekProFun          |
DekProImp          |
DekWieProImp       |
get-object ( Identyfikator ) |
WyrObi ; WyrObi
```

W tym kroku transformacji zostały opuszczone nazwy konstruktorów oraz nawiasy związane ze średnikiem. Uwagi w sprawie legalności takich konstrukcji w rozdziałach 6.2.2 i 7.8.2.

deo : DefObi =


```

set-object Identyfikator as WyrObi tes-object |
DefObi ; DefObi

```

```

prw: Przywołanie =
  Identyfikator          |
  Przywołanie ; Przywołanie

```

```

prx : ProgramPrefiksowany =
  begin prog
    call objects
      Przywołanie
    end call
      Preambuła ;
      Instrukcja
  end prog          |
  Program

```

9.7 Programowanie walidacyjne w Lingua-3

Wszystkie konstrukcje i rozważania związane z programowaniem walidacyjnym w **Lingua-2** przenoszą się bez istotnych zmian na **Lingua-3**, gdyż semantycznie przywołania można postrzekać jako składowe preambuły.

10 Obiekty zewnętrzne — zarys idei

Przedstawiony w niniejszej książce denotacyjny model języków programowania można wykorzystać w sytuacjach, w których użytkownikowi **Lingua** udostępniamy zasoby — czyli zbiory danych i narzędzia — leżące poza standardowym „wyposażeniem” naszego języka takie jak na przykład bazy danych SQL, arkusze kalkulacyjne Excela, systemy skryptowe HTML, czy też systemy sterowania urządzeniami fizycznymi. W każdym z takich przypadków rozbudowany **Lingua** stawałby się środowiskiem programistycznym pozwalającym wykorzystywać całe bogactwo naszego języka, a ponadto sięgać do zasobów danych oraz narzędzi aplikacji zewnętrznych.

Technicznie rzecz ujmując, w każdym takim przypadku trzeba denotacyjny model **Lingua** rozbudować o narzędzia wybranej aplikacji, co oznacza, że dla tej ostatniej należy zbudować model denotacyjny. Jest to niewątpliwie najtrudniejsza część zadania, gdyż dostępne podręczniki aplikacji są najczęściej niejasne i niekompletne, a nierzadko też wewnętrznie sprzeczne. Jednakże, gdy taki model raz powstanie, możemy na jego gruncie zbudować napisać lepszy i krótszy podręcznik danej aplikacji przeznaczony dla czytelnika naszej książki, któremu nie trzeba już tłumaczyć wszystkiego od początku używając do tego języka gazetowej prozy.

Rozwiązania umożliwiające na sięganie w programach jakiegoś języka do zasobów zewnętrznych wobec niego, nie są oczywiście niczym nowym. Jednakże w takich rozwiązaniach język programowania pozwala jedynie na uruchamianie z wnętrza programu akcji zewnętrznego silnika, podczas gdy jego autorzy nie biorą żadnej odpowiedzialności za efekt tych akcji. Trudno zresztą byłoby się spodziewać, że wezmą odpowiedzialność za coś, co nie jest ich, skoro nie biorą jej nawet za dzieło własne.

W **Lingua** sytuacja jest jednak inna. Skoro nasz język ma udostępniać programiście narzędzia budowania programów poprawnych, to musimy brać odpowiedzialność nie tylko za nasze programy, ale też ze uruchamianie z ich wnętrza aplikacje zewnętrzne. To z kolei prowadzi do konieczności zbudowania dla tych aplikacji modeli denotacyjnych, a następnie wywodzących się z nich implementacji. Co więc w tej sytuacji oznacza, że rozbudowujemy **Lingua** o jakiś obiekt zewnętrzny? W jaki sposób możemy maksymalnie wykorzystać to co już jest w danej aplikacji nie rezygnując jednocześnie z walidacyjnych cech naszego języka? Wydaje się, że możemy oczekiwać zrealizowania trzech zadań:

- A. takiego rozbudowania struktur danych **Lingua**, aby obejmowały struktury aplikacji zewnętrznej, np. bazy danych standardu SQL, co zostanie pokazane w rozdziale 12
- B. stworzenie asortymentu konstruktorów nowych struktur danych w taki sposób, aby były one „dostatecznie bliskie” konstruktorom zewnętrznej aplikacji,
- C. dodanie własnych konstruktorów, których nie ma w aplikacji, a które uznalibyśmy za pożyteczne.

W przypadku zadania B proponuję „dostateczną bliskość” rozumieć jako zbieżność wyników w sytuacjach typowych. Na przykład w niektórych implementacjach SQL argumentami

arytmetycznego dodawania mogą być słowa, które „przypominają liczby” (por. [39], s. 753). Wtedy implementacja „domyśla się”, że dane słowo należy potraktować jak liczbę, co powoduje, że np. $2 + ab3 = 5$. W takiej sytuacji „nasze” dodawanie pokrywałoby się z dodawaniem SQL-owskim dla argumentów liczbowych, generując jednak komunikat błędu w pozostałych sytuacjach. Wyrażając to dokładniej, **Lingua** pozwalający na przetwarzanie danych strukturalnych zewnętrznej aplikacji, np. baz danych SQL czy arkuszy Excela, dopuszczałby operowanie na danych importowanych z innych aplikacji, ale wykorzystywałbym do tego własne konstruktory jedynie pokrywające się z konstruktorami danej aplikacji w „typowych” sytuacjach.

W rozdziałach 11 i 12 podejmuję próbę przedstawienia na przykładzie SQL pewnej metody wzbogacania **Lingua** o obiekty zewnętrzne, na którą składają się cztery etapy:

1. Opisanie aplikacji w sposób na wpół formalny, tj. z zachowaniem dyscypliny pojęciowej obowiązującej w **Lingua**, ale bez posługiwania się formalizmem modelu denotacyjnego. Na tym etapie identyfikowane są niejednoznaczności i luki źródłowych podręczników aplikacji, by oddzielić to, co jest w niej jasne i pewne od tego co nie do końca określone lub różnie realizowane w różnych implementacjach.
2. Zbudowanie denotacyjnego modelu dla tego podzbioru systemu, który dało się opisać w punkcie pierwszym.
3. Wbudowanie obiektu zewnętrznego do **Lingua**, to znaczy wyposażenie tego ostatniego w środki językowe pozwalające na korzystanie z mechanizmów obiektu.
4. Rozbudowanie walidującej warstwy **Lingua** o narzędzia pozwalające na formułowanie tez obejmujących nowy obiekt.

Pierwsze dwa kroki stanowią etap, na którym możemy podjąć decyzję o ograniczeniu obiektu zewnętrznego do pewnego jego podzbioru w taki sposób, aby to co pozostanie:

- dawało się opisać w sposób formalny, a więc jednoznaczny,
- było dostatecznie proste, by pozwalało na stworzenie sensownych reguł dowodzenia poprawności.

W praktyce krok pierwszy i drugi będą się przeplatać, gdyż dopiero na etapie budowania opisu denotacyjnego można ustalić, które z konstrukcji aplikacji dają się jednoznacznie określić.

11 Relacyjne bazy danych intuicyjnie

11.1 Uwagi wstępne

Rozdział 12 poświęcam rozszerzeniu **Lingua-3**, które obejmie wybrane narzędzia bazodanowe typowe dla standardu *Structured Query Language* (SQL). Postąpię jednak inaczej niż twórcy typowych środowisk programistycznych dla SQL zwanych w języku angielskim *Application Programming Interface* w skrócie API ([39]) lub też *Call Level Interface* (CLI)¹⁰⁴.

API zostały stworzone na bazie języków programowania C, PHP, Perl i Python, a CLI na bazie ANSI C, C#, VB.NET, Java, Pascal i Fortran¹⁰⁵. Każde z tych środowisk to pewien uniwersalny język programowania wyposażony w mechanizmy uruchamiania procedur jakiegoś zewnętrznego silnika bazy danych. W przypadku **Lingua-SQL** postąpię inaczej. Zostanie on wyposażony we własny silnik, którego procedury będą semantycznie odpowiadały typowym procedurom SQL-owym, ale zostaną zdefiniowane denotacyjnie, a w przyszłości (mam nadzieję) zaimplementowane niezależnie. Takie ujęcie jest niezbędne dla zaproponowania wiarygodnych reguł tworzenia programów poprawnych. Więcej na ten temat na początku rozdziału 12.1.

Wiedzę o SQL i związanych z nim aplikacjach czerpię z kilku źródeł, gdyż dla większości pojęć nie udaje się ustalić ich znaczenia na podstawie jednego podręcznika. Książka Lecha Banachowskiego [7] zawiera teoriorelacyjny model dla *Relacyjnych Baz Danych* i dość czytelny opis standardu SQL, choć wiele zagadnień jest w niej pominiętych (np. definicje trójwartościowych operatorów boolowskich), a inne jedynie naszkicowane. Na drugim krańcu skali znajduje się ponad tysiącstronicowe dzieło Paula DuBois [39], które łączy w sobie rozwlekłość z niepełnością i brakiem jasności. To wręcz podręcznikowy przykład, jak nie należy pisać podręczników. Z tej książki przytaczam niektóre sformułowania, aby pokazać czytelnikowi z jakimi problemami trzeba się będzie liczyć budując bazodanowy obiekt dla **Lingua** w wymiarze praktycznym. Pomiędzy oboma ekstremami, ale stanowczo bliżej Dubois, leżą cztery inne książki [41], [46], [57], [62] również pełne luk i niejasności. Oczywiście we wszystkich tych źródłach brak jest choćby zrębu denotacyjnego modelu dla SQL lub którejkolwiek z pochodnym mu aplikacji.

Ponieważ wszystkie te książki zostały wydane już jakiś czas temu, niektóre z opisanych tam szczegółów technicznym mogą wyglądać dziś nieco inaczej. Nie ma to jednak szczególnego znaczenia, gdyż i tak wszystkie bazodanowe konstruktory SQL zostaną zdefiniowane niezależnie. Oczywiście dbać będę o to, aby były one możliwie bliskie standardowi SQL oraz — co najważniejsze — aby dawały się stosować do baz danych stworzonych istniejące aplikacje.

Od czytelnika nie oczekuję znajomości SQL. Rozdział 11 poświęcam więc nieformalnemu opisowi wybranych konstrukcji występujących w SQL-owych API, by zbudować podstawowe intuicje konieczne do stworzenia własnego silnika baz danych. Niekiedy wprowadzam też

¹⁰⁴ CLI odwołuje się do standardu ANSI SQL ([62] s. 359).

¹⁰⁵ Na żadnej z tych list nie wymieniono aplikacji Access, gdyż jest na dostępna jedynie z fabrycznie wbudowanym językiem Basic Access. Jest to więc API dla Access zbudowane na Microsoft Basic.

pojęcia niewystępujące w SQL, ale niezbędne do uporządkowania opisów występujących w nim mechanizmów. Będę je nazywał *pojęciami własnymi*.

Modelowi denotacyjnemu dla **Lingua-SQL** jest w całości poświęcony rozdział 12.

11.2 Dane proste

W znanych mi podręcznikach SQL jeden tylko rodzaj danych strukturalnych występuje *explicit* i są to *tabele bazodanowe* (rozdział 11.3). *Implicit* występują też wiersze tabel i bazy danych (rekordy tabel), ale tego się trzeba w zasadzie domyśleć. Poza danymi strukturalnymi występują też cztery rodzaje (ale wiele typów) *danych prostych* (pojęcie własne)¹⁰⁶, które mogą być zapisywane w polach tabel. Dane proste stanowią jeden z najbardziej nieustandaryzowany obszarów SQL-owych aplikacji. Rodzaje i typy tych danych, a także repertuary związanych z nimi operatorów różnią się nie tylko pomiędzy aplikacjami, ale też pomiędzy ich poszczególnymi implementacjami. Być może dlatego właśnie we wszystkich znanym mi podręcznikach SQL jest to najgorzej opisany mechanizm — pełen luk, niejasności i wewnętrznych sprzeczności.

W niniejszym rozdziale oprę się w dużej mierze¹⁰⁷ na podręczniku [62], którego autorzy deklarują zgodność przedstawianej przez nich wersji SQL ze standardem ANSI SQL-2011¹⁰⁸. Do składni SQL stosuję krój Arial Narrow.

Tabele bazodanowe mogą przechowywać dane czterech rodzajów, z których każdy, poza wartościami logicznymi, dzieli się na wiele typów:

- **Liczby** najprościej rzecz ujmując dzielą się na trzy podrodzaje: całkowite, dziesiętne i zmiennopozycyjne, z których każdy obejmuje wiele typów różniących się zakresami wielkości (jajzami), np. INTEGER, SMALLINT, BIGINT, czy też DECIMAL(p, s), gdzie p (precyzja) oznacza dopuszczalną liczbę cyfr danej wartości, a s (skala), to liczba cyfr stojących po przecinku. Z kolei FLOAT(p) to liczby zmiennopozycyjne, dla których (por. [62] s.40) akceptowalna jest dowolna precyzja i skala (to czemu służy parametr p?), natomiast według [46] s.176 precyzja, którą określa parametr, jest minimalną wartością, ale konkretna implementacja może ją zwiększyć.
- **Wartości logiczne** są przyjmowane jak w trójwartościowym rachunku zdań Kleene’ego i w [62] są nazwane TRUE, FALSE oraz NULL natomiast w [39] są to 1, 0 i NULL. Niekiedy też (np. [46]) w rolę NULL gra UNKNOWN.
- **Łańcuchy** są w zasadzie słowami w naszym sensie, a dzielą się na typy, podobnie jak liczby, ze względu na dopuszczalną liczbę znaków. Np. CHARACTER(n) to typ słów o długości n. Jeżeli w pole tabeli tego typu wpisujemy słowo krótsze, to jest ono na końcu uzupełniane spacjami ([62] s. 37). Typ łańcucha o zmiennej długości ograniczonej z góry przez parametr n nosi nazwę (wg. [62]) CHARACTER VARYING(n), a typ łańcucha o nieograniczonej długości (cokolwiek to oznacza!) to BLOB. Występują też łańcuchy binarne, oraz TEXT — łańcuchy tekstowe.

¹⁰⁶ Do tej pory do danych prostych zaliczałem jedynie dane boolowskie, liczby i słowa, teraz ich rodzajów będzie więcej.

¹⁰⁷ Głównie, a nie całkowicie, bo ten podręcznik również zawiera luki.

¹⁰⁸ ANSI to skrót od American National Standard Institute, a SQL-2011 to standard SQL przyjęty przez ANSI w grudniu 2011 roku.

- **Czasy** są krotkami trzech typów: DATE — (rok, miesiąc, dzień), TIME — (godzina, minuta, sekunda), DATETIME — (rok, miesiąc, dzień, godzina, minuta, sekunda)

Choć nie jest to nigdzie wyraźnie powiedziane, w [62] i innych źródłach można się między wierszami doczytać, że wszystkie rodzaje danych obejmują pseudodaną NULL grającą w zasadzie rolę błędu abstrakcyjnego. Większość też konstruktorów, poza boolowskimi, wydaje się być transparentna względem NULL.

Konstruktory danych prostych można podzielić na pięć wymienionych poniżej grup¹⁰⁹.

1. Operatory arytmetyczne: +, −, *, /.
2. Operatory łańcuchowe: CONCAT, UPPER, LOWER, SUBSTR, LENGTH.
3. Operatory czasu: GETDATE, DAYNAME, DAYOFMONTH,
4. Predykaty podstawowe: =, <>, <, <=, >, >=, IS NULL, BETWEEN, LIKE.
5. Spójniki logiczne: NOT, OR, AND.

Pierwsza grupy operatorów wydaje się dość oczywista, okazuje się jednak, że jest tak tylko w sytuacjach typowych. $2+3=5$, ale gdy wchodzimy w obszar wykraczający poza normalność, np. próbujemy dodać liczbę do łańcucha lub dodać do siebie dwie liczby, których suma przekraczałaby maksymalną wartość dopuszczalną, zaczynają się niejasności. Źródło [62] w ogóle nie odnosi się do takich sytuacji, a w [39] s. 786 możemy przeczytać:

Nie dostarczając (...) funkcjom prawidłowych wartości (...), nie należy oczekiwać rozsądnych wyników.

W innym miejscu tego samego podręcznika (s. 754) czytamy:

*(...) wyrażenia zawierające duże liczby mogą przekroczyć zakres obliczeń 64-bitowych, dając nieprzewidywalne wyniki (podkreślenie moje)*¹¹⁰.

Na szczególne podkreślenie zasługuje również fakt, że w definicjach operatorów arytmetycznych nie korzysta się z wartości NULL, której można by użyć jako błędu abstrakcyjnego. W to miejsce przyjęto rozwiązanie najgorsze z możliwych: zamiast błędu pojawia się „nieprzewidywalny wynik” czyli wynik niezgodny z zasadami arytmetyki, co jednak nie przerywa biegu programu. Program wykonuje się do końca, ale daje fałszywy wynik, o czym użytkownik nie zostaje poinformowany!

Szczególnie dużo niejasności jest też związanych z zasadami domyślności przy konwersji typów. Na przykład ([39], s. 753) podana jest następująca reguła dotycząca operacji dodawania w zastosowaniu do argumentów łańcuchowych:

... + nie jest operatorem łączenia tekstów, jak w niektórych językach programowania. Zamiast tego przed wykonaniem operacji arytmetycznej łańcuchy tekstowe są przekształcane na liczby. Łańcuchy, które nie wyglądają na liczby (podkreślenie moje), są przekształcane na 0.

Ta zasada została zilustrowana następującymi przykładami:

`'43bc' + '21d' = 64`

¹⁰⁹ Opisy konstruktorów z grup 1. do 4. czerpię z [62] (s. 129 i 180), a z grup 5. i 6. z [46] s. 191 i 201. Używam jednak własnej terminologii i własnej systematyki pojęć.

¹¹⁰ Zwróćmy uwagę na pojęciowe niechlujstwo sformułowania „wyrażenia mogą przekroczyć zakres obliczeń”. Powinno być oczywiście napisane, że wyliczane wartości tych wyrażeń mogą nie zmieścić się w dopuszczalnym zakresie.

'abc' + 'def' = 0

Nie wyjaśniono jednak, czy np. '43ab2c' „wygląda na liczbę”, a gdyby tak było, to czy zostanie przekształcony na 43, czy na 432? Nie wyjaśniono też, czy te reguły stosują się do pozostałych operatorów arytmetycznych.

Na szczęście [62] traktuje sprawę konwersji bardziej odpowiedzialnie — choć nadal całkowicie nieformalnie — wprowadzając jedynie cztery typy konwersji:

1. łańcuchy na liczby,
2. liczby na łańcuchy,
3. łańcuchy na daty i czasy,
4. daty i czasy na łańcuchy.

Operatory łańcuchowe stwarzają mniej okazji do niejasności, jednak nadal definiowane są w zasadzie jedynie dla sytuacji typowych. Np. nie udało mi się znaleźć informacji, co się stanie, gdy będziemy próbowali połączyć dwa łańcuchy dające w wyniku łańcuch przekraczający dopuszczalną długość.

Operatory czasu to kolejny obszar rozbieżności pomiędzy różnymi aplikacjami SQL zarówno co do składni, jak i rodzaju operatorów. Ponieważ jednak w każdej konkretnej sytuacji ich definicje sformułować jest łatwo, nie będę ich tu omawiał.

Predykaty to pojęcia typologicznie wieloznaczne w tym sensie, że w większości stosują się do wszystkich czterech rodzajów danych. Np. operatorów = oraz BETWEEN można użyć zarówno do liczb, jak i łańcuchów, a pewnie też i do dat. Najczęściej ich definicje są bardzo ogólne. Np. ([62] s. 130) czytamy:

Jeżeli w zapytaniu używamy operatora (=), porównywane wartości muszą być identyczne, w przeciwnym przypadku warunek nie będzie spełniony.

Nie wyjaśniono jaką wartość będzie miało wyrażenie boolowskie $12 = abc$ — pp czy ee.

Operator BETWEEN bierze trzy argumenty i sprawdza, czy pierwszy leży pomiędzy drugim i trzecim w jakimś domyślnym dla nich porządku.

Operator LIKE bierze dwa argumenty łańcuchowe i sprawdza, czy pierwszy pasuje do wzorca opisanego przez drugi. Wzorce opisujemy za pomocą liter i cyfr oraz dwóch *symboli wieloznacznych*:

% — dowolny ciąg znaków być może pusty,

_ — dowolny znak

Jedynym źródłem, w którym udało mi się znaleźć pełne definicje spójników logicznych jest [46], gdzie na stronie 191 są one zdefiniowane metodą tabelkową, w której przyjąłem notację meta-software (Rys. 11.2-1). Jak widzimy, są to operatory rachunku zdań Kleene'ego (patrz rozdział 2.9). Kolorem zaznaczyłem te sytuacje, które odróżniają operatory Kleene'ego od McCarthy'owskich. Jak pamiętamy, rachunek zdań Kleene'ego jest nieimplementowalny, o ile w języku mogą istnieć nieskończone obliczenia wyrażeń. Jak zobaczymy dalej, w przypadku SQL takiego zagrożenia niema.

OR	pp	ff	bb	AND	pp	ff	bb	NOT	
pp	pp	pp	pp	pp	pp	ff	bb	pp	ff
ff	pp	ff	bb	ff	ff	ff	ff	ff	pp
bb	pp	bb	bb	bb	bb	ff	bb	bb	bb

Rys. 11.2-1 Spójniki logiczne w SQL

Pomimo istnienia spójnika NOT, dla wszystkich predykatów podstawowych wprowadza się ich wersje zanegowane jako odrębne operatory, np. NOT NULL oraz NOT BETWEEN.

W przypadku wszystkich grup operatorów, za wyjątkiem spójników logicznych, mamy sytuację typową dla podręczników systemów informatycznych — dopóki poruszamy się w obszarze przypadków standardowych, wszystko jest jasne, jednak gdy wykraczamy poza nie, na ogół nie wiemy już, co może się zdarzyć. Z dużą pewnością możemy jedynie przewidywać, że w każdej implementacji spotka nas inna niespodzianka.

Na koniec jeszcze jedna uwaga. Otóż w typowych językach programowania dane proste mogą być przypisywane zmiennym jako ich wartości. Jednakże w SQL jest inaczej. Jak zobaczymy dalej, dane proste mogą być przypisywane jedynie polom tabel.

11.3 Tworzenie tabel

Centralnym pojęciem standardu SQL jest *tabela bazodanowa*. Na gruncie naszego modelu można uznać, że tabele są krotkami rekordów przechowujących dane proste. Zgodnie z powszechnie przyjętą terminologią zawarte w tabelach rekordy nazywam *wierszami*, atrybuty tych rekordów — *nazwami kolumn*, a przecięcia kolumn i wierszy — *polami*.

Tabele w SQL — a dokładniej odpowiadające im dane utypowane, czyli wartości w sensie zdefiniowanym w rozdziale 5.3.1 — są (chyba?) jednym z rodzajów wartości, które mogą być przypisywane zmiennym. Zmienne o wartościach będących tabelami będę dalej nazywał *zmiennymi tabelowymi* (pojęcie własne). Do zadeklarowania zmiennej tabelowej służy konstruktor CREATE TABLE, który nazwie zmiennej (identyfikatorowi) przypisuje *typ tabelowy* oraz (należy się tego domyślać) jakiś rodzaj *tabeli pustej* (pojęcie własne) cokolwiek to oznacza.

Typ tabelowy to korpus rekordowy uzupełniony o dodatkowe cechy atrybutów, które można podzielić na dwie grupy — jarzma w sensie opisanym w rozdziale 5.2.4 oraz dane domyślne, których obecność wykracza nieco poza nasz model typów, daje się jednak do niego wbudować (rozdział 12.2). Oto przykład dwóch takich deklaracji (podaję z drobnymi zmianami za [5] s.14)¹¹¹:

¹¹¹ W rozdziale 12 będę się odwoływał do tego przykładu, a także do przykładów pochodzących z [5]. W obu przypadkach zachowuję oryginalną notację, gdzie Number(p) oznacza typ liczb całkowitych o liczbie cyfr równej p, natomiast Number(p, s) oznacza typ liczb dziesiętnych o całkowitej liczbie cyfr równej p, a w tym po przecinku liczbie cyfr równej s. Z kolei Varchar(n) oznacza typ łańcuchów o długości nieprzekraczające n.

```
CREATE TABLE Działy
```

```
(  
  Id_działu      Number(3) PRIMARY KEY,  
  Nazwa_działu   Varchar(20) NOT NULL      UNIQUE  
  Miasto         Varchar(50)  
);
```

```
CREATE TABLE Pracownicy
```

```
(  
  Id_pracownika  Number(6) PRIMARY KEY,  
  Nazwisko       Varchar(20) NOT NULL,  
  Stanowisko     Varchar(9)  DEFAULT NULL,  
  Kierownik      Number(6) ,  
  Data_zatrudnienia Date,  
  Pensja         Number(8,2),  
  Premia         Number(8,2),  
  Id_działu      Number(3) REFERENCES Działy,  
  CHECK (Premia < Pensja)  
)
```

Zastosowana w tym przykładzie tabulacja służy pokazaniu struktury deklaracji:

- w pierwszej kolumnie deklaracji stoją nazwy kolumn tabeli, czyli wspólne atrybuty rekordów składających się na tabelę,
- w pozostałych kolumnach deklaracji (może być ich więcej niż trzy) są zapisane informacje o cechach danych stojącym w kolumnach; w naszym modelu będą one wyrażane za pomocą korpusów i jarzm,
- w ostatnim wierszu drugiej deklaracji stoi warunek określający oczekiwany związek pomiędzy wartościami pól Pensja oraz Premia w każdym z wierszy przyszłej tabeli; premia nie może być większa od zarobków; w terminologii wprowadzonej w rozdziale 5.2.4 to jarzmo powstaje przez użycie konstruktora *wszystkie-tri*.

Kolumny deklaracji począwszy od drugiej, a także ostatni jej wiersz, określają tzw. *więzy spójności* lub *ograniczenia*. Ich znaczenia są następujące:

1. Number(3) — Typ danych dopuszczalnych w danej kolumnie.
2. DEFAULT — Wartość domyślna
3. NOT NULL — Wszystkie pola w kolumnie muszą zostać wypełnione, tj. żadne z nich nie może być NULL. Próba naruszenia tego ograniczenia powinna spowodować wstrzymanie operacji i wygenerowanie sygnału błędu.

4. UNIQUE — W kolumnie nie mogą wystąpić dwie identyczne wartości, co oznacza że, gdyby przy wstawianiu wartości do tabeli miało dojść do naruszenia tego warunku, silnik bazy wstrzyma operację i wygeneruje sygnał błędu.
5. PRIMARY KEY — Ta kolumna stanowi *klucz główny* tabeli. Klucz główny musi być *kluczem jednoznacznym*, czyli takim, że wartość klucza w wierszu jednoznacznie wskazuje ten wiersz. Klucz główny może być wyznaczony przez więcej niż jedną kolumnę. Silnik bazy dba o zachowanie jednoznaczności klucza w analogiczny sposób jak dla ograniczenia UNIQUE.
6. REFERENCES Działy — Pole *Id_działu* w tabeli *Pracownicy* jest powiązane z identycznie nazwanym polem w tabeli *Działy*. Aby ta część deklaracji została zaakceptowana przez silnik, tabela *Działy* musi być wcześniej zadeklarowana. Powiązania są wykorzystywane przy modyfikowaniu tabel i zadawaniu zapytań (o czym dalej).
7. CHECK(Premia < Pensja) — W przypadku dodawania nowego wiersza do tabeli lub modyfikacji istniejącego wiersza przez zmianę wartości jednego lub obu z dwóch wskazanych pól (kolumn), silnik będzie wstrzymywał operację i generował sygnał błędu, gdyby po modyfikacji tabeli warunek *Premia < Pensja* miałby nie być spełniony.

Jak widzimy z tego przykładu, przy deklarowaniu zmiennej tabelowej definiujemy jej typ, a więc korpus i jarzmo¹¹². Ten typ obejmuje pięć grup cech przyszej tabeli:

1. nazwy kolumn,
2. typy wartości stojących w polach kolumny, np. *Number(6)*,
3. ograniczenia dotyczące wartości stojących w kolumnach takie jak PRIMARY KEY, NOT NULL czy też UNIQUE,
4. warunki określające związki pomiędzy wartościami stojącymi w każdym z wierszy, np. CHECK(Premia < Pensja),
5. powiązania wskazanych kolumn tabeli z kolumnami innych tabel, np. REFERENCES Działy.

Jak już pisałem, cechy kolumn odpowiadające grupom od 2. do 5. będę nazywał więzami spójności¹¹³ danej tabeli. Ograniczenia wszystkich tabel w bazie danych wchodzi w skład *więzów spójności bazy danych*. Nie są to jednak jedyne dopuszczalne rodzaje więzów spójności. Innym przykładem mogą być warunki zrównoważenia bilansu banku przy wykonywaniu operacji bankowych (przykład w rozdziale 11.5.)

Niestety nie udało mi się ustalić czym klucz typu UNIQUE różni się od PRIMARY KEY poza tym, że ten ostatni może być tylko jeden. Jak podano w [62] s.62 ... *to klucz główny określa kolejność danych w tabeli i pozwala łączyć powiązane ze sobą tabele*. W innym jednak miejscu (s. 216) ci sami autorzy o złączeniu zewnętrznym piszą tak: *Łączy ono dwie tabele za pomocą wspólnej kolumny, która w każdej z tabel spełnia zazwyczaj (podkreślenie moje) rolę klucza głównego*. A skoro „zazwyczaj” to pewnie nie zawsze. Być może więc jedyną cechą odróżniającą klucz główny od kluczy jednoznacznych jest to, że przy wyświetlaniu tabeli, jest ona uporządkowana według tego klucza.

Na koniec kilka refleksji dotyczących użytego na początku rozdziału pojęcia *tabeli pustej*, która jest przypisywana zmiennej tabelowej przez operację CREATE TABLE. W literaturze

¹¹² Wydaje się, że w SQL brak jest mechanizmów, które pozwalałyby na zdefiniowanie typu tabelowego inaczej niż w trakcie deklarowania zmiennej.

¹¹³ Niektórzy autorzy nazywają je *więzami integralności* lub *ograniczeniami*.

bazodanowej takie pojęcie oczywiście nie występuje, nie udało mi się też znaleźć informacji jaki rodzaj danej jest przypisywany zmiennej tabelowej w chwili jej deklarowania. Można się tego jedynie domyślać zważywszy, że po zadeklarowaniu zmiennej tabelowej można się odnosić do przypisanej jej tabeli np. dodając do niej kolumny i wiersze. Czym jednak jest tabela pusta? Czy może jest to tabela jednowierszowa, której pola nie są wypełnione? Raczej nie, o czym świadczy przykład podany na początku niniejszego rozdziału, gdzie od tworzonej tabeli Działy wymaga się, aby pole Nazwa_działu nie było puste. Jednakże deklaracja tabeli nie wskazuje żadnej danej, która miałaby stać w tym polu — nie wskazuje wartości domyślnej — co powoduje, że w chwili deklarowania zmiennej tabelowej Działy nie możemy jej przypisać niczego, co przypominałoby tabelę. Zatem *tabela pusta*, to nie tabela, ale coś na kształt „zapowiedzi przyszłej tabeli” lub „miejsce na nią” w pamięci komputera. W budowanym dalej modelu denotacyjnym będzie to więc znana nam już pseudowartość Ω .

11.4 Relacje nadrzędności tabel

Intuicyjnie rzecz biorąc *relacje* to związki pomiędzy tabelami, które pozwalają wykonywać operacje tabelowe na kilku powiązanych z sobą tabelach. W terminologii **Lingua-A** można by powiedzieć, że odpowiadają one jarzmom nakładanym na rekordy tabel, czyli bazy danych.

Mechanizm ustanawiania relacji pomiędzy tabelami występuje w aplikacjach SQL w kilku wersjach. Wszystkie one odpowiadają pewnej wspólnej idei, choć eksploatują ją każda na swój sposób. W niniejszym rozdziale postaram się opisać tę ideę wprowadzając kilka pojęć własnych.

Rozważmy tablice Działy i Pracownicy zdefiniowane w rozdziale 11.3. W tabeli Pracownicy występuje kolumna *ld_działu*, która opisuje przynależność pracownika do działu. W deklarację tego pola wpisano ograniczenie REFERENCES Działy co ma wskazywać, że w tabeli Działy znajdziemy informacje o dziale, w którym jest zatrudniony dany pracownik. Zamiast w rekord każdego pracownika wpisywać wszystkie dane zatrudniającego go działu, podajemy jedynie identyfikator tego działu pozwalający na odszukanie danych działu w tabeli Działy. Jednak by taka konstrukcja miała praktyczny sens, nasze dwie tabele muszą spełniać trzy warunki:

1. w obu tabelach musi występować kolumna *ld_działu*,
2. każdy identyfikator działu występujący w tabeli Pracownicy musi występować w tabeli Działy,
3. w tabeli Działy atrybut *ld_działu* i musi być kluczem jednoznacznym.

Jeżeli te trzy własności są spełnione, to powiemy, że:

atrybut ld_działu wiąże tabele Działy i Pracownicy relacją typu (1-W) (jeden do wielu).

Każdemu działowi może być przypisany dowolnie liczny, w tym pusty, zbiór pracowników, natomiast każdemu pracownikowi musi być przypisany dokładnie jeden dział.

W tej parze Działy jest *tabelą nadrzędną*, a Pracownicy — *tabelą podrzędną*. W niektórych aplikacjach mówi się też o *tabeli podstawowej* i *powiązanej*. Atrybut *ld_działu* jest w tabeli Działy *kluczem głównym*, a w tabeli Pracownicy *kluczem obcym*.

Jeżeli wiersz pracownika WP i wiersz działu WD mają w polu *ld_działu* tę samą wartość, to powiemy, że wiersz WP *wskazuje na wiersz WD* (pojęcie własne).

Przez (1-W) oznaczę teraz relacją trójczłonową będącą podzbiorem iloczynu kartezjańskiego trzech dziedzin:

$$(1-W) \subset \text{Tabela} \times \text{Atrybut} \times \text{Tabela}$$

taką, że jeżeli (tab-1, atr, tab-2) : (1-W), to tab-1 jest nadrzędna dla tab-2 przy kluczu głównym atr.

Trójka uporządkowana (Działy, Id_działu, Pracownicy) z naszego przykładu stanowi więc element takiej relacji. W tym przypadku atrybut Id_działu nazwiemy *kluczem łączącym* nasze tabele.

Zauważmy teraz, że zachodząca pomiędzy tymi tabelami relacja (1-W) może zostać zepsuta przy modyfikowaniu jednej lub obu tabel, np. gdy:

- z tabeli Działy usuniemy rekord wskazywany przez jakiś rekord z tabeli Pracownik,
- do tabeli Pracownik wstawimy rekord z identyfikatorem działu nieistniejącego w tabeli Działy lub gdy w pole Id_działu wstawimy NULL,
- w którejkolwiek z tabel przemianujemy atrybut Id_działu,
- do tabeli Działy wstawimy rekord z występującym już w tej tabeli identyfikatorem działu, tj. gdy zepsujemy jednoznaczność klucza Id_działu.

Fakt, że dwie tabele pozostają w relacji (1-W) można wykorzystywać na wiele sposobów zarówno przy generowaniu raportów jak i przy tworzeniu nowych tabel. Jednakże sprawdzanie przy każdej takiej okazji, czy argumenty operacji pozostają pomiędzy sobą w relacji (1-W) byłoby mało praktyczne. Znacznie lepiej jest z góry zadeklarować, że tak ma być, a następnie zadbać o to, aby silnik bazy danych nie pozwolił na naruszenie tego warunku.

W naszym przykładzie deklaracja o utrzymaniu związku (1-W) pomiędzy tabelami Działy i Pracownicy jest zawarta w deklaracji odpowiadających im zmiennych tabelowych¹¹⁴:

- w deklaracji tabeli Działy atrybut Id_działu został zadeklarowany jako PRIMARY KEY; przypominam, że każdy klucz główny musi być jednoznaczny,
- w deklaracji tabeli Pracownicy atrybut Id_działu otrzymał ograniczenie REFERENCES Działy.

Fakt ustanowienia relacji (1-W) pomiędzy tabelami ma swoje konsekwencje przy wykonywaniu operacji na połączonych tabelach. Na przykład:

- Nie daje się wprowadzić do bazy pracownika zatrudnionego w dziale nieopisanym w tabeli działów. Silnik bazy wymusza opisanie działu w pierwszej kolejności.
- Rekordu działu nie daje się usunąć z bazy działu, dopóki są w nim zatrudnieni jacyś pracownicy. Alternatywne rozwiązanie może być takie, że usunięcie rekordu działu powoduje automatyczne usunięcie wszystkich związanych z nim rekordów pracowników.
- Można zażądać wygenerowania tabeli o trzech polach łączących informacje z obu połączonych tabel, np.: Nazwisko, Nazwa_działu, Miasto.

Szczególnym przypadkiem relacji (1-W), jest relacja (1-1), gdzie dla każdego rekordu w tablicy nadrzędnej istnieje co najwyżej jeden rekord w tabeli podrzędnej. Zauważmy, że „co najwyżej jeden”, a nie „dokładnie jeden”, co oznacza, że relacja (1-1) nie musi być symetryczna. Nadal więc jedna z tabel takiej pary jest nadrzędna, a druga podrzędna.

¹¹⁴ Niektóre aplikacje, np. Access (por. [57]) oferują środowisko graficzne pozwalające na wygodne deklarowanie powiązań pomiędzy tabelami, które są widoczne jako krawędzie grafów o węzłach będących tabelami.

Dla sformalizowania rozważań o relacjach podrzędności wprowadzam pojęcie *grafu podrzędności*, którym będzie każdy skończony, być może pusty, zbiór trójek identyfikatorów:

$\text{grp} : \text{GraPod} = \text{SkPod}(\text{Identyfikator} \times \text{Identyfikator} \times \text{Identyfikator})$

Elementy tego zbioru nazywam *krawędziami grafu podrzędności*. Intuicyjnie każda krawędź (ide-p , ide , ide-n) wskazuje w bazie danych relację nadrzędności, która ma miejsce pomiędzy tabelami o nazwach ide-p (tabela podrzędna) oraz ide-n (tabela nadrzędna) względem atrybutu ide .

11.5 Instrukcje modyfikowania tabel

Zadeklarowane tabele, a także tabele udostępnione (patrz rozdział 12.7.6.11), można modyfikować za pomocą instrukcji, w których występują pewne specyficzne konstruktory. Poniżej przykłady najważniejszych z nich:

Wstawienie do tabeli nowej kolumny:

```
ALTER TABLE Pracownicy
    ADD COLUMN Pesel CHAR(11) DEFAULT NULL
```

Dodajemy kolumnę do tabeli i podajemy jej wartość domyślną.

Usuwanie kolumny z tabeli

```
ALTER TABLE Działy
    DROP COLUMN Id_działu CASCADE (lub RESTRICT)
```

Wykonanie tej instrukcji z opcją `CASCADE` spowoduje kaskadowe usunięcie z bazy danych wszystkich obiektów (tablic, perspektyw,...), które odwołują się do tej kolumny. W przypadku użycia `RESTRICT` operacja się nie wykona, jeżeli w bazie istnieją takie odwołania.

Zauważmy, że instrukcje z grupy `ALTER TABLE` dokonują nie tylko modyfikacji treści tabeli, ale też zmieniają jej typ. Obok wariantów podanych powyżej istnieje jeszcze kilka innych ([46] s. 49):

- `ALTER COLUMN` — zmiana typu kolumny przez użycie `SET DEFAULT` lub `DROP DEFAULT` dodające lub usuwające deklarację wartości domyślnej.
- `ADD` — dodanie ograniczenia do wskazanej kolumny.
- `DROP CONSTRAINT` — usunięcie ograniczenia ze wskazanej kolumny. Dla tego polecenia należy zadeklarować jedną z opcji `RESTRICT` lub `CASCADE`.

Drugi rodzaj instrukcji modyfikujących tabele zmienia zawartość tabeli pozostawiając jej typ nienaruszonym. Oto typowe przykłady:

Wstawienie nowego wiersza (rekordu):

```
INSERT INTO Działy
    VALUES (095, 'Narzędziownia', 'Katowice')
```

Tę instrukcję można też zapisać w wersji, w której nazwy kolumn są wymienione explicite (por. [39] str.73)

```
INSERT INTO Działy (Id_działu, Nazwa_działu, Miasto)
    VALUES (095, 'Narzędziownia', 'Katowice')
```


Modyfikowanie wszystkich danych wskazanej kolumny. Np. zwiększenie zarobków wszystkim sprzedawcom o 10%:

```
UPDATE Pracownicy
SET Pensja = Pensja * 1,1
WHERE Stanowisko = 'sprzedawca'
```

Usuwanie wierszy spełniających określony warunek. Np. usuwanie pracowników, którzy nie mają określonego stanowiska:

```
DELETE FROM Pracownicy
WHERE Stanowisko IS NULL
```

Szczególną sytuacją stanowią przypadki, gdy usuwamy z tablicy wiersz, w którym występuje klucz główny będący kluczem obcym w tablicy podrzędnej, np.

```
DELETE FROM Działy
WHERE Nazwa_działu = 'produkcja'
```

Jeżeli w tabeli podrzędnej Pracownicy klucz `Id_działu` jest — jak w naszym przypadku — kluczem obcym oraz istnieją wiersze, które wskazują na wiersz usuwany z tabeli Działy, to operacja się nie wykona i zostanie wygenerowany sygnał błędu, gdyż w przeciwnym przypadku nastąpiłoby zaburzenie więzów spójności bazy. Jeżeli jednak wykonamy instrukcję w wersji:

```
DELETE FROM Działy
WHERE Nazwa_działu = 'produkcja' CASCADE
```

to operacja się wykona, a wraz z nią z tablicy Pracownicy zostaną usunięte wszystkie wiersze wskazujące na wiersz usuwany z tabeli Działy¹¹⁵.

11.6 Transakcje

Transakcją nazywa się w SQL sekwencję instrukcji ujętą (lub nie) w jakieś (różnie z tym bywa) nawiasy, na przykład `BEGIN TRANSACTION` i `COMMIT TRANSACTION`¹¹⁶. Nad wykonaniem transakcji czuwa mechanizm, który będę dalej nazywał *mechanizmem przywracania* (nazwa własna)¹¹⁷, powodujący, że jeżeli jakaś instrukcja transakcji generuje sygnał błędu, np. dlatego, że jej wykonanie naruszałoby więzy spójność lub też dlatego, że danej instrukcji nie daje się wykonać (np. odwołanie do nieistniejącego rekordu), to stan bazy wraca do stanu wskazanego *explicite* w transakcji lub — jeżeli taki stan nie został wskazany — do stanu sprzed wykonania transakcji.

¹¹⁵ Jest zastanawiającą niekonsekwencją twórców SQL, że w tym przypadku nie wprowadzono opcji `RESTRICT` jak w przypadku usuwania kolumn, a uczynioną ją jedynie domyślną, gdy `CASCADE` nie wystąpi. Nota bene w żadnym z powoływanych przeze mnie podręczników nie znalazłem w definicjach `DELETE` wyjaśnienia, co się dzieje, gdy zabraknie w nich opcji `CASCADE`. W niektórych znalazłem jedynie informację, że jeżeli w czasie wykonywania transakcji (rozdział 11.6) więzy spójności miałyby zostać naruszone, to polecenie się nie wykona i pojawi się komunikat błędu.

¹¹⁶ Te nawiasy mogą mieć różną leksykę w różnych aplikacjach, a w niektórych podręcznikach w ogóle się o nich nie wspomina. Tu powołuję się na podręcznik Bena Forty [41] s.175, który pisze, że „niektóre aplikacje wymagają, by jednoznacznie określić początek i koniec bloku transakcji”. Microsoft SQL Server wskazuje przyjętą przez mnie składnię Forty jako obowiązującą dla aplikacji.

¹¹⁷ Muszę zastrzec, że mechanizm przywracania opisany jest we wszystkich znanych mi podręcznikach w sposób wyjątkowo niejasny i niepełny, to więc co piszę na jego temat jest jedynie wynikiem moich przypuszczeń.

Do programistycznego sterowania mechanizmem przywracania służy pięć kolejnych typów instrukcji, które mogą występować (jak się wydaje) jedynie w transakcjach:

SAVEPOINT	— zapisz punkt zachowania
RELEASE SAVEPOINT	— usuń punkt zachowania
ROLLBACK	— odwołaj transakcję
IF	— warunkowa aktywizacja przywrócenia
COMMIT TRANSACTION	— zatwierdź transakcję,

Instrukcja

SAVEPOINT nazwa-punktu-zachowania

powoduje przypisanie aktualnego stanu całej bazy do tymczasowej zmiennej *nazwa-punktu-zachowania*. Instrukcja

RELEASE SAVEPOINT nazwa-punktu-zachowania

usuwa wskazany punkt zachowania. Instrukcja

ROLLBACK nazwa-punktu-zachowania

powoduje przywrócenie bazy do stanu przypisanego zmiennej *nazwa-punktu-zachowania* z jednoczesnym usunięciem tej zmiennej z pamięci. Ta instrukcja może występować również bez parametru¹¹⁸, co jak się wydaje oznacza, że wtedy powrót następuje do stanu początkowego dla wykonania transakcji. To by z kolei sugerowało, że rozpoczęcie wykonywania bloku transakcyjnego zaczyna się od wykonania niewidocznej w składni (domyślnej) transakcji SAVEPOINT przypisującej aktualny stan bazy jakiejś zmiennej systemowej. Wydaje się też, że wykonanie ROLLBACK powinno przerywać bieg programu i emitować komunikat błędu.

Aby wykonanie ROLLBACK móc programowo uzależnić od pojawienia się komunikatu błędu, wprowadza się konstruktor warunkowy IF. Informację na ten temat znalazłem jedynie u Bena Forty [41] (s. 179) i jedynie w postaci następującego przykładu:

IF @@ERROR <> 0 ROLLBACK *nazwa-punktu-zachowania*

Jak wyjaśniono tamże, @@ERROR jest zmienną systemową, która jako wartości przyjmuje 0, gdy nie pojawi się błąd, lub (jak sądzę) komunikat błędu w przeciwnym przypadku.

Z tego przykładu nie wynika, choć można by się tego spodziewać, że w warunku instrukcji IF mogłoby stać wyrażenie boolowskie postaci:

@@ERROR = komunikat-błędu

co pozwalałoby uzależnić wybór punktu zachowania od treści komunikatu.

Wykonanie instrukcji COMMIT powoduje zapisanie na stałe wyniku wykonania transakcji i usunięcie z pamięci wszystkich zadeklarowanych wcześniej punktów zachowania.

Na przykład, w bazie danych przechowującej informacje o klientach banku, transakcja przełania 1000 zł z konta klienta o numerze 1250 na konto klienta o numerze 1260 może wyglądać następująco:

BEGIN TRANSACTION

SAVEPOINT początek

¹¹⁸ Tak jest właśnie opisana w większości znanych mi podręczników.

```
UPDATE Konta
```

```
SET Saldo = Saldo - 1000
```

```
WHERE Id_klienta = 1250 ;
```

```
IF @@ERROR <> 0 ROLLBACK początek ;
```

```
UPDATE Konta
```

```
SET Saldo = Saldo + 1000
```

```
WHERE Id_klienta = 1260 ;
```

```
IF @@ERROR <> 0 ROLLBACK początek
```

```
COMMIT TRANSACTION
```

Pierwsze wykonanie ROLLBACK będzie miało miejsce wtedy, gdy w bazie brak klienta o numerze 1250 lub gdy jego saldo jest niższe od 1000 zł. Drugie ROLLBACK uaktywni się wtedy, gdy pierwsze nie zostanie uaktywnione, ale w bazie nie ma klienta o numerze 1260.

Zauważmy, że po wykonaniu pierwszej instrukcji UPDATE suma depozytów na kontach nie jest równa bilansowej sumie złożonych depozytów. Mamy sytuację zwaną *niespójnością bazy* lub *naruszeniem więzów spójności*. Druga instrukcja usuwa tę niespójność, jednak gdyby nie mogła się wykonać, z braku klienta o numerze 1260, to wynikiem transakcji stałaby się baza niespójna. Potrzebne jest więc drugie wywołania operacji ROLLBACK.

11.7 Zapytania

Zapytania służą do pozyskiwania informacji z bazy danych, tj. z jednej lub większej grupy tabel. Wykonanie zapytania powoduje wygenerowanie tabeli i wyświetlenie jej na monitorze. Zapytania są tworzone za pomocą różnych wariantów operatora SELECT. Oto kilka charakterystycznych przykładów:

Wybór z tablicy wskazanych kolumn:

```
SELECT Nazwisko, Pensja, Stanowisko  
FROM Pracownicy
```

W wyniku tego zapytania na monitorze zostaje wyświetlona tablica o trzech kolumnach wymienionych w wierszu SELECT.

Wybór kolumn połączony z filtrowanym wyborem wierszy:

```
SELECT Nazwisko, Pensja, Stanowisko  
FROM Pracownicy  
WHERE Id_działu = 10
```

W klauzuli WHERE mogą występować wyrażenia boolowskie zawierające operatory na danych prostych opisane w rozdziałach 11.2 oraz 12.3.

Zapytania można też składać z innych zapytań za pomocą konstruktorów, które Banachowski [7] nazywa „operatorami algebraicznymi na zapytaniach”¹¹⁹. Mogą też sięgać do więcej niż jednej tabeli. Na przykład:

```
SELECT Id_działu
FROM Działy
EXCEPT
SELECT Id_działu
FROM Pracownicy
```

Powyższe zapytanie generuje jednokolumnową tabelę tych identyfikatorów działów, które występują w tablicy Działy, ale nie występują w tabeli Pracownicy. Są to więc numery działów, które nie zatrudniają żadnych pracowników.

Wśród konstruktorów z tej grupy mamy również UNION, UNION ALL (suma obejmująca powtórzenia) oraz INTERSECT.

Szczególne typy stanowią zapytania sięgające do więcej niż jednej tabeli. Mówimy o nich, że posługują się one *złączeniami tabel*. Oto przykład takiego zapytania dokonującego wyboru z dwóch tabel: Pracownicy oraz Działy.

```
SELECT Id_pracownika, Nazwisko, Id_działu
FROM Pracownicy, Działy
WHERE Pracownicy.Id_działu = Działy.Id_działu AND Działy.Miasto = 'Warszawa'
```

To zapytanie generuje tablicę o trzech kolumnach, której każdy wiersz zawiera identyfikator pracownika, jego nazwisko i nazwę zatrudniającego go działu. Występujący w klauzuli WHERE warunek nazywa się *predykatem złączenia*. W tym przypadku powoduje on utworzenie jedynie takich wierszy, które odpowiadają pracownikom zatrudnionym działach położonych w Warszawie.

W klauzuli WHERE mogą występować zarówno wyrażenia boolowskie wykorzystujące predykaty podstawowe algebry danych (rozdział 11.2), np.:

```
SELECT Id_pracownika, Nazwisko, Pensja
FROM Pracownicy
WHERE Pensja > 1000 AND Pensja <= 2000
```

jak i predykaty teoriomnogościowe opisane w rozdziale 12.3 (w SQL noszą one nazwy IN, ALL, EXISTS, SOME i ANY przy czym SOME i ANY są synonimami). Na przykład zapytanie:

```
SELECT Id_pracownika, Nazwisko, Stanowisko, Pensja
FROM Pracownicy
WHERE Stanowisko IN ('kasjer', 'sprzedawca', 'kierownik').
```

generują tabelę obejmującą kasjerów, sprzedawców i kierowników. Zapytanie:

```
SELECT Id_pracownika, Nazwisko, Stanowisko, Pensja
```

¹¹⁹ Banachowski ma tu na myśli algebrę relacji stanowiącą model dla relacyjnych baz danych. Nie chodzi tu jednak o relacje pomiędzy tablicami (rozdział 11.4), ale o spojrzenia na tablice, jako na zbiory krotek jednakowej długości, a więc relacje n-członowe.

```

FROM Pracownicy
WHERE Pensja > ALL (
  SELECT Pensja
    FROM Pracownicy
    WHERE Stanowisko = 'kasjer' )

```

generuje tabelę obejmującą pracowników o pensjach przewyższających pensje wszystkich kasjerów. W tym przypadku mamy do czynienia z *zapytaniem zagnieżdżonym*, gdzie wewnętrzne SELECT generuje kolumnę zawierającą pensje wszystkich kasjerów. Oznaczmy:

pep : PenPra — zbiór wartości stojących w kolumnie Pensja tabeli Pracownicy,
 pek : PenKas — podzbiór PenPra zawierający pensje kasjerów,
 pwk : PenWyżKas — podzbiór PenPra zawierający pensje wyższe od pensji kasjerów.

Wtedy:

$$\text{PenWyżKas} = \{ \text{pep} \mid \text{pep} : \text{PenPra} \text{ oraz } (\forall \text{pek} : \text{PenKas}) \text{pep} > \text{pek} \}$$

czyli, zgodnie z notacją wprowadzoną w rozdziale 12.3:

$$\text{PenWyżKas} = \{ \text{pep} \mid \text{pep} : \text{PenPra} \text{ oraz } \underline{\text{all}}.(\text{PenKas}, >).\text{pep} = \text{pp} \}$$

gdzie $>$ oznacza predykat porównywania liczb przyjmujący wartość *bb*, gdy którykolwiek z jego argumentów nie jest liczbą.

Z transparentności predykatu $>$ wynika oczywiście, że zbiór *PenWyżKas* będzie zawierał jedynie liczby, choć oczywiście może być pusty. W szczególności będzie pusty, gdy *PenKas* zawiera choćby jedną nie-liczbę.

W żadnym ze źródeł nie udało mi się znaleźć informacji, czy w przypadku błędu pojawiającego się w wyniku ewaluacji $\text{pep} > \text{pek}$ nastąpi przerwanie wykonania zapytania i sygnalizacja błędu, czy też zapytanie się wykona i wygeneruje jakąś tabelę, być może pustą.

Rozważmy teraz zapytanie powstające z poprzedniego, gdy *ALL* zamienimy na *EXISTS*, tj. generujące tabelę pracowników o zarobkach wyższych od choć jednego kasjera¹²⁰:

```

SELECT Id_pracownika, Nazwisko, Stanowisko, Pensja
FROM Pracownicy
WHERE Pensja > EXISTS (
  SELECT Pensja
    FROM Pracownicy
    WHERE Stanowisko = 'kasjer' )

```

Oznaczmy:

pwn : PenWyżNka — pensje wyższe od niektórych pensji kasjerów.

Wtedy:

$$\text{PenWyżNka} = \{ \text{pep} \mid \text{pep} : \text{PenPra} \text{ oraz } (\exists \text{pek} : \text{PenKas}) \text{pep} > \text{pek} \}$$

¹²⁰ W tym przypadku stosuję składnię, co do której nie jestem pewien, czy jest dopuszczalna. Przyjąłem ją jednak, by uzyskać podobieństwo do przykładu z *ALL*, którego składnię (choć nie sam przykład) znalazłem w [62] s.139.

czyli, zgodnie z notacją wprowadzoną w rozdziale 12.3:

$\text{PenWyżNka} = \{\text{pep} \mid \text{pep} : \text{PenPra} \text{ oraz } \text{exists.}(\text{PenKas}, >).\text{pep} = \text{pp}\}$

W tym przypadku, w odróżnieniu od poprzedniego, jeżeli PenKas zawiera nie-liczby, to zbiór PenWyżNka nie musi być pusty.

Zauważmy teraz, że ilekroć ewaluacja wyrażenia $\text{pep} > \text{pek}$ dla jakiegoś pek , wygeneruje błąd, to:

$\text{exists.}(\text{PenKas}, >).\text{pep} = \text{ff}$

Gdybyśmy jednak w naszym zapytaniu zamienili EXISTS na SOME, to mogłaby się pojawić wartość bb . To oczywiście nie ma wpływu na wygenerowaną przez zapytanie tablicę, ale ma wpływ na generowanie komunikatów błędów, o ile oczywiście jest generowany.

Kwantyfikatory mogą też występować w kontekście łączenia tablic. Poniższe zapytanie generuje tabelę działów, w których zatrudniono co najmniej jednego pracownika.

```
SELECT Id_działu
FROM Działy
WHERE Id_działu = EXISTS (
    SELECT Id_działu
    FROM Pracownicy)
```

Jak pisałem w rozdziale 11.2, dla każdego operatora prostego wprowadza się jego wersję zanegowaną, np. $=$ i $\triangleleft$, LIKE i NOT LIKE itd. Podobnie wprowadza się NOT IN. Jednakże w przypadku kwantyfikatorów mnogościowych znalazłem jedynie NOT EXISTS i jedynie w [62] s.147 oraz [39] s.242. Rzecz jasna, w żadnym z tych źródeł, nie odniesiono się do przypadku, gdy kwantyfikowane wyrażenie przyjmuje wartość bb .

Z denotacyjnego punktu widzenia zapytania można traktować jako wyrażenia, bo one jedynie generują pewną wartość (tabelę), ale nie zmieniają stanu pamięci.

11.8 Funkcje agregujące

Funkcje agregujące SUM, MAX, MIN, AVG przyjmują jako argument jednokolumnową tabelę liczb będącą wynikiem zapytania i zwracają odpowiedni wynik. Jeżeli tabela jest pusta, to zwracają NULL ([46] s.148).

Funkcja COUNT przyjmuje dowolną tabelę jednokolumnową i zwraca liczbę jej wierszy pomijając te, w których stoi NULL. Jej szczególna wersja COUNT(*) przyjmuje dowolną tabelę i zlicza wszystkie jej wiersze łącznie z duplikatami ([62] s. 155).

11.9 Perspektywy

Zapytań można używać jednorazowo — np. do wyświetlenia wyniku zapytania na monitorze — lub też do deklarowania procedur, by odwoływać się później do nich przez ich nazwy. Takie procedury nazywają się w SQL *perspektywami* (ang. *view*). A oto przykład deklaracji perspektywy:

```
CREATE VIEW Urzędnicy
(Id_pracownika, Imię, Nazwisko, Pensja)
```

```
AS SELECT Id_pracownika, Imię, Nazwisko, Pensja
FROM Pracownicy
WHERE Stanowisko = 'urzędnik'
```

Ta perspektywa o nazwie *Urzędnicy* tworzy czterokolumnową tabelę przez wybór kolumn z tabeli *Pracownicy* i wierszy z tejże tablicy dla których w kolumnie *Stanowisko* stoi 'urzędnik'.

Ponieważ perspektywy są procedurami, nie mają swoich odpowiedników na poziomie składni (por. rozdział 7.1.3). W składni występują jedynie *deklaracje perspektyw* `CREATE VIEW` oraz ich *przywoływania* (pojęcie własne) przez przypisaną im nazwę (identyfikator).

Do perspektyw możemy się odwoływać w zapytaniach w identyczny sposób, jak do tabel, przy czym wywołanie perspektywy powoduje jej wykonanie na stanie bazy z czasu wywołania, a nie deklaracji. W podręcznikach SQL perspektywy są więc niekiedy określane jako *wirtualne tabele*. Perspektywy możemy też wywoływać w instrukcjach modyfikujących lub tworzących tabele. Rozważmy następującą deklarację perspektywy:

```
CREATE VIEW Pracownicy_sprzedaży
AS SELECT * FROM Pracownicy
WHERE Id_działu = 20
```

W tej deklaracji gwiazdka "*" w klauzuli wyboru oznacza, że wybieramy wszystkie kolumny, natomiast 20 to numer działu sprzedaży. Odwołując się do perspektywy *Pracownicy_sprzedaży* możemy napisać instrukcję modyfikującą tabelę *Pracownicy* w ten sposób, że wszystkim sprzedawcom podnosimy wynagrodzenie o 10%:

```
UPDATE Pracownicy_sprzedaży
SET Pensja = Pensja * 1,1.
```

W przypadku użycia perspektyw do modyfikowania tabel każdy system RBD wprowadza pewne ograniczenia. Np. w MySQL w klauzuli `SELECT` mogą się pojawić jedynie nazwy kolumn.

Szczególny przypadek perspektyw stanowią *perspektywy z opcją sprawdzania*, które wymuszają sprawdzenie warunku, gdy są używane w instrukcjach. Banachowski [7] podaje przykład takiej perspektywy:

```
CREATE VIEW Pracownicy_na_urlopie_bezpłatnym
AS SELECT *
FROM Pracownicy
WHERE Pensja = 0 OR Pensja IS NULL
WITH CHECK OPTION
```

Jeżeli takie perspektywy użylibyśmy w instrukcji

```
UPDATE Pracownicy_na_urlopie_bezpłatnym
SET Pensja = 1000
WHERE Nazwisko = 'Kowalski'
```

to ona się nie wykona, o ile pensja Kowalskiego jest 0 lub NULL.

11.10 Kursory

Kursory są narzędziami służącymi do pobierania z tabel pojedynczych wierszy i przypisywania ich zmiennym. Mechanizm kursorów pozwala na przetwarzanie danych zawartych w bazach za pomocą programów pisanych w językach interfejsu użytkownika (API lub CLI; por. rozdział 11.1). Kursor wskazuje konkretną tabelę, z której będą pobierane wiersze oraz aktualnie dostępny wiersz tej tabeli. Tabele wskazywane przez kursor są definiowane za pomocą zapytań. W zasadzie więc powinniśmy mówić nie o kursorze jako takim, ale o *kursorze tabeli*, a może wręcz o *kursorze zapytania*.

Kursor tworzymy za pomocą *deklaracji kursora*, która nazwie kursora (identyfikatorowi) przypisuje kursor i jest postaci¹²¹:

```
DECLARE nazwa_kursora IS  
SELECT ...
```

Po zadeklarowaniu kursora nie jest on jeszcze gotowy do użytku. Aby go takim uczynić należy wykonać instrukcję otwarcia:

```
OPEN nazwa_kursora.
```

Ta operacja powoduje wykonanie zawartej w deklaracji operacji SELECT, a więc wygenerowanie tablicy, oraz — jak się domyślam, choć nigdzie się tego nie doczytałem — ustawienie tzw. *uchwyty kursora* przed pierwszym wierszem wygenerowanej tabeli. Operacja pobierania danych z tabeli jest postaci:

```
FETCH NEXT nazwa_kursora INTO zmienna
```

Słowo NEXT oznacza pobranie wiersza kolejnego w stosunku do wskazywanego przez uchwyt i przesunięcie uchwytu o jeden wiersz niżej. Stąd właśnie moje przypuszczenie, że OPEN ustawia uchwyt przed pierwszym wierszem.

Instrukcja FETCH NEXT jest zwykle umieszczona w pętli jakiegoś programu, co wiąże się z faktem, że po dojściu do ostatniego wiersza tabeli przesunięcia uchwytu nie daje się dokonać. W tej sprawie w [62] s. 353 znalazłem jedynie taki komentarz:

W każdej implementacji baz danych uchwyt kursorów są zaimplementowane w nieco inny sposób, ale każdy umożliwia poprawne zamknięcie kursora bez niepotrzebnego generowania błędów.

Zamknięcia kursora dokonujemy za pomocą instrukcji

```
CLOSE nazwa_kursora
```

która powoduje likwidację kursora, co najczęściej oznacza jedynie tyle, że przestaje on być otwarty, ale jego deklaracja „pozostaje w mocy”.

11.11 Środowisko klient-serwer

Do tej pory mówiliśmy o systemach SQL zakładając milcząco, że użytkownik systemu ma bazę danych do swojej wyłącznej dyspozycji. Tak jednak najczęściej nie jest. Użytkowników jest zwykle wielu, trzeba więc sprawić, by nie wchodzili sobie w drogę. Temu służą narzędzia udzielania i odbierania dostępu oraz zakładania blokad. Oto przykład ogólnego schematu instrukcji zakładającej blokadę na wskazaną tablicę:

¹²¹ Składnia deklaracji zależy od wersji aplikacji. Tu posłużyłem się wersją ORACLE (za [62] s. 352).


```
LOCK TABLE nazwa_tabeli  
IN [SHARE | EXCLUSIVE]  
[NOWAIT]
```

gdzie w nawiasach kwadratowych stoją opcje o następujących znaczeniach:

- SHARE — blokada obejmująca wszystkich użytkowników
- EXCLUSIVE — blokada obejmująca wszystkich użytkowników za wyjątkiem zakładającego blokadę,
- NOWAIT — nie czekać na założenie blokady, o ile nie można jej w danej chwili założyć.

Zdjęcie blokady następuje przez wykonanie instrukcji COMMIT lub ROLLBACK. Przykładem instrukcji nadającej wybrane uprawnienia wskazanemu użytkownikowi może być:

```
GRANT SELECT, UPDATE (Pensja)  
ON Pracownicy  
TO Księgowa
```

Ta instrukcja nadaje użytkownikowi Księgowa uprawnienia do wykonywania operacji SELECT i UPDATE w tablicy Pracownicy.

Te aspekty SQL różnią się pomiędzy jego różnymi aplikacjami, a przy tym są techniczne dość proste do opisanie, nie będę się więc nimi dalej zajmować. Nie uwzględnę ich też **Lingua-SQL**. Oczywiście dla potrzeb praktycznych zastosowań trzeba będzie tę lukę wypełnić.

12 Lingua-SQL

12.1 Ogólne założenia dotyczące modelu

Jak już wyjaśniłem w rozdziale 11.1, budując **Lingua-SQL** przyjmuję założenie, że ten język będzie zawierał własne implementacje SQL-owych operacji na bazach danych. Oznacza to, że model denotacji dla **Lingua-3** należy rozbudować:

1. o nowe dziedziny danych odpowiadające pojęciom związanym z SQL-owymi bazami danych takimi jak wiersze tabel oraz specyficzne dla SQL dane proste,
2. o definicje konstruktorów i ich pochodnych na tych dziedzinach, tzn. konstruktorów

W niniejszej książce nie podejmę oczywiście trudu zdefiniowania praktycznego repertuaru narzędzi SQL-owych, gdyż moim celem jest przedstawienie metody budowania denotacyjnego modelu baz danych, a nie zbudowanie użytecznego środowiska programowania. Mam jednak nadzieję, że na gruncie opisanego dalej modelu, będzie można podjąć w przyszłości trud stworzenia takiego środowiska.

12.2 Kompozyty

12.2.1 Dane, korpusy i kompozyty SQL-owe

W dotychczas zdefiniowanych wersjach **Lingua** wartościami były pary składające się z kompozytu i transferu spełnianego przez ten kompozyt. W modelu dla SQL będzie tak w przypadku wartości niosących dane proste, wiersze i tabele, natomiast wartościami bazodanowymi będą rekordy wartości tabelowych uzupełnione o grafy relacji podrzędności (rozdział 12.6). By nie komplikować zbytnio naszego modelu, wartości bazodanowe nie będą jednak przypisywane identyfikatorom. Będą im więc przypisywane jedynie wartości wierszowe i tabelowe, natomiast jedynymi konstruktorami odwołującymi się do pojęcia bazy danych będą (rozdział 12.7.6.11):

- aktywizacja i archiwizacja bazy oraz
- deklarowanie i odwoływanie relacji podrzędności tabel.

Zadbam też o to, aby SQL-owe dane nie mieszały się z danymi pochodzącymi z **Lingua-3** w tym sensie, że listy, rekordy i tablice nie będą mogły nieść wierszy, tabel i baz danych, a pola tabel nie będą mogły zawierać list, rekordów i tablic. Dopuszczę natomiast, aby rozszerzona dziedzina danych prostych w **Lingua-SQL** była dostępna dla konstruktorów list, rekordów i tablic.

Jako SQL-owe dane proste rozszerzające dotychczasowy repertuar przyjmuję dane związane z kalendarzem i zegarem:

dat	: Data	= Rok x Miesiąc x Dzień
cza	: Czas	= Godzina x Minuta x Sekunda

dcz : DataCzas = Data x Czas

gdzie:

rok : Rok = {0,...,9999} (to tylko przykład)

mie : Miesiąc = {1,...,12}

dzi : Dzień = {1,...,31}

god : Godzina = {0,...,23}

min : Minuta = {0,...,59}

sek : Sekunda = {0,...,59}

Ponieważ dane proste grają w **Lingua-SQL** szczególną rolę, wprowadzam dziedzinę danych prostych:

dap : DanProsta = Boolowska | Liczba | Słowo | Data | Czas | DataCzas | { Θ }

oraz zakładam, że wszystkie konstruktory z poziomu **Lingua-3**, które przyjmują jako argumenty dane proste, a więc np. dodawanie atrybutu do rekordu, zostają w naturalny sposób rozszerzone na ich nową dziedzinę.

Dla uwzględnienia w naszym modelu wierszy i tabel zawierających pola puste, zakładam istnienie *danej pustej*, którą oznaczam przez Θ ¹²². Będzie ona mogła stać w polach tabel, a więc być przypisywana atrybutom wierszy, ale nigdy nie wystąpi jako wartość wyrażenia, ani też nie będzie przypisywana zmiennym danologicznym.

W związku ze zwiększeniem repertuaru danych prostych może się też zwiększyć repertuar konstruktorów tych danych, np. o dodawanie liczby do daty. Tych konstruktorów nie będę definiował *explicite* przyjmując, że ich zbiór stanowi parametr modelu. Zakładam jednak, że nie obejmują one żadnych SQL-owych „reguł domyślności” w rodzaju dodawania słowa do liczby, o których była mowa w rozdziale 11.2.

Jeżeli chodzi o podkategorie liczb w rodzaju INTEGER, SMALLINT, BIGINT, DECIMAL(p, s), czy też słów CHARACTER(n), CHARACTER VARYING(n), BLOB, to pojawią się one dopiero na poziomie jarzm.

Na nową dziedzinę danych prostych rozszerzam relację *równe* wprowadzoną w rozdziale 5.2.1.

Zgodnie z wcześniejszą zapowiedzią wprowadzam dwa nowe rodzaje danych strukturalnych:

wie : Wiersz = Identyfikator $\Rightarrow$ DanProsta

tab : Tabela = Wiersz^{c*}

Zwracam uwagę na zasadnicze różnice pomiędzy pojęciami *tablicy* z **Lingua-3** oraz *tabeli* z **Lingua-SQL**:

- tablica jest mappingiem odwzorowującym liczby w dowolne dane, a w tym w dowolnie złożone dane strukturalne **Lingua-A**,
- tabela jest krotką wierszy (być może pustą) przechowujących jedynie dane proste.

¹²² Zwracam uwagę na różnicę pomiędzy wprowadzoną w rozdziale 5.3.1 pseudodaną Ω , która jest przypisywana zmiennej danologicznej w chwili jej deklarowania, a daną pustą Θ , która może być przypisywana atrybutom wierszy i reprezentuje puste pole wiersza lub tabeli.

Na poziomie równań dziedzinowych tabele zawierają wiersze, z których każdy może być nie tylko innej długości, ale też zawierać inne atrybuty. Oczywiście takie tabele nie będą należeć do osiągalnej części nośnika tabel w algebrze kompozytów. Tabelę o pustej krotce wierszy nazwę *tabelą pustą*.

Bazy danych nie występują na poziomie danych, a jedynie na poziomie wartości, co zostanie wyjaśnione w rozdziale 12.6.

Analogicznie jak w **Lingua-A** wszystkim danym SQL-owym przypisuję odpowiadające im korpusy. Korpusy danych prostych definiuję jako jednoelementowe krotki słów:

$$\text{kor} : \text{KorProsty} = \{('boolowska'), ('liczba'), ('słowo'), ('data'), ('czas'), ('data-czas')\},$$

natomiast korpusy nowych danych strukturalnych definiuję w sposób następujący:

$$\text{kor} : \text{KorWiersz} = \{ 'Wq' \} \times \text{RekWiersz}$$

$$\text{rwi} : \text{RekWiersz} = \text{Identyfikator} \Rightarrow \text{KorProsty}$$

$$\text{kor} : \text{KorTabela} = \{ 'Tq' \} \times \text{Wiersz} \times \text{KorWiersz}$$

Jak można się domyśleć z tych definicji, kompozyty wierszy występujących w tabelach osiągalnych będą miały, podobnie jak w przypadku list, wspólny korpus. Występujący w korpusie tabelowym wiersz niesie informacje o danych domyślnych dla kolumn tabeli. Jego lista atrybutów musi pokrywać się z listą atrybutów korpusu wierszowego, o co zadbamy na poziomie konstruktorów korpusów.

Zakładam, że dziedzina **KorpusB** zostaje powiększona o nowe korpusy proste, korpusy wierszowe i tabelowe.

Na korpusy wierszowe funkcję **KLAN-Kr** z **Lingua-A** rozszerzam w sposób oczywisty. W przypadku korpusów tabelowych zakładam nie tylko, że każdy wiersz tabeli należącej do klanu ma mieć odpowiednią strukturę rekordową, ale też, że w polach, dla których wartość domyślna jest niepusta, nie stoi wartość pusta. Oczywiście w tych polach nie musi stać wartość domyślna. Wartości domyślne będą wykorzystane przy dokładaniu do tabeli nowego wiersza (rozdział 12.2.6) lub nowej kolumny (rozdział 12.2.7).

Zakładam, że tabela pusta należy do klanu każdego korpusu tabelowego.

Na nowe korpusy i kompozyty rozszerzam w sposób oczywisty zdefiniowaną w rozdziałach 5.2.2 i 5.2.3 funkcję **rodzaj**.

Dotychczasową dziedzinę kompozytów **KompozytB** również w sposób oczywisty rozszerzam na kompozyty związane z nowymi danymi prostymi, a także z danymi wierszowymi i tabelowymi. Dodatkowo wprowadzam też pomocniczą dziedzinę kompozytów prostych:

$$\text{kom} : \text{KomProsty} =$$

$$\{(\text{dan}, \text{kor}) \mid (\text{dan}, \text{kor}) : \text{KompozytB} \text{ oraz } \text{kor} : \text{KorProsty}\}$$

Zakładam, że dla każdego korpusu prostego **kor**:

$$\Theta : \text{KLAN-Kr.kor}$$

a zatem (Θ, kor) jest kompozytem dla każdego prostego **kor**.

12.2.2 Relacje podrzędności tabel

Relacje *podrzędności tabel* opisują związki, jakie mogą mieć miejsce pomiędzy dwiema tabelami. Rozważmy więc dwie tabele A i B i niech *ide* będzie atrybutem występującym w obu tabelach. Przez *A.ide* oraz *B.ide* oznaczę odpowiednie kolumny w naszych tabelach.

Powiemy, że tabela A, że jest *podrzedną wobec B przez ide*, lub alternatywnie że tabela B jest *nadrzedną wobec A przez ide*, co zapiszemy jako:

A pod[ide] B

gdy są spełnione trzy warunki:

1. kolumna o atrybucie *ide* występuje w obu tabelach,
2. kolumna *B.ide* nie zawiera powtórzeń, co oznacza, że każdy jej element jednoznacznie wskazuje wiersz, do którego należy,
3. kolumna *A.ide* zawiera jedynie dane występujące w kolumnie *B.ide*, co wraz z warunkiem 2. oznacza, że każdy wiersz tabeli A jednoznacznie wskazuje na pewien wiersz tabeli B.

Na Rys. 12.2-1 widzimy taki przykład, gdzie zachodzi relacja

Pracownicy pod[Dział] Działy

Atrybut *ide* nazwiemy *indykatorem podrzędności* (pojęcie własne) dla A i B. W każdej tabeli może być więcej niż jeden taki indykator. Kolumnę *B.ide* nazwiemy *kolumną nadrzedną* dla *A.ide*, a *A.ide* — *kolumną podrzedną* dla *B.ide*.

A: Pracownicy		B: Działy	
Nazwisko	Dział	Dział	Miasto
Kowalski	Narzędziownia	Narzędziownia	Kraków
Malinowski	Narzędziownia	Lakiernia	Poznań
Jankowski	Lakiernia	Blacharnia	Bydgoszcz

Rys. 12.2-1 Pracownicy *podrzedna dla Działy przez Dział*

Jeżeli w kolumnie nadrzędnej *B.ide* występuje element, który w kolumnie podrzędnej *A.ide* występuje więcej niż raz (więcej niż jeden pracownik jest zatrudniony w narzędziowni, jak w naszym przykładzie), to mówimy że relacja nadrzędności jest typu (1-W) (jeden do wielu). W przeciwnym przypadku mówimy, że jest typu (1-1). Zauważmy, że w obu przypadkach mogą istnieć w kolumnie nadrzędnej elementy, które nie występują w kolumnie podrzędnej (działy w których nikt nie pracuje). To powoduje, że relacja typu (1-1) nie musi być symetryczna.

Zauważmy, że relacja nadrzędności odnosi się do tabel, a nie do kompozytów tabelowych, co oznacza, że do rozstrzygnięcia, czy dwie tabele są powiązane tą relacją, nie musimy sięgać do ich korpusów. Można jednak łatwo rozszerzyć ją na kompozyty zakładając definicyjną równoważność:

(tab-1, kor-1) pod[ide] (tab-2, kor-2) wtw (def) tab-1 pod[ide] tab-2

Jak łatwo stwierdzić przy modyfikowaniu tabeli można zepsuć dotyczące ją podrzędności (bo może być ich wiele) w następujących przypadkach:

- A. gdy usuniemy kolumnę związaną z indykatoem nadrzędności lub podrzędności (warunek 1.),
- B. gdy do tabeli nadrzędnej B dodamy wiersz, który wprowadzi powtórzenie do kolumny indykatora (warunek 2.),
- C. gdy z tabeli nadrzędnej B usuniemy wiersz, na który wskazywał jakiś wiersz tabeli podrzędnej A (warunek 3.),
- D. gdy do tabeli podrzędnej A dodamy wiersz, który do kolumny indykatora wprowadzi daną nie występującą w kolumnie indykatora tabeli nadrzędnej B (warunek 3.).

Stany programów operujących na bazach danych muszą nieść informację dotyczącą zadeklarowanych relacji podrzędności pomiędzy tabelami. Aby uwzględnić ten mechanizm w naszym modelu wprowadzam pojęcie *grafu podrzędności* (pojęcie własne), którym będzie skończony zbiór trójek identyfikatorów:

$grp : \text{GrafPod} = \text{Pod.}(\text{Identyfikator} \times \text{Identyfikator} \times \text{Identyfikator})^{123}$

Każdą należącą do grafu krotkę (*ide-p*, *ide*, *ide-n*) nazwę *krawędzią grafu podrzędności*, gdzie *ide-p* i *ide-n* pełnią rolę wierzchołków grafu, a *ide* — rolę etykiety krawędzi. W kontekście zadanego stanu pamięci każda krawędź tego grafu wyraża fakt, że pomiędzy niesionymi przez stan tabelami o nazwach *ide-p* i *ide-n* zachodzi relacja podrzędności, dla której *ide* jest indykatoem podrzędności.

O grafach podrzędności zakładam jedynie, że dla każdej krawędzi *ide-p* $\neq$ *ide-n*, jednakże grafy podrzędności mogą zawierać cykle. Zauważmy też, że z jednego wierzchołka może wychodzić wiele krawędzi (jedna tabela może mieć wiele tabel nadrzędnych), jak też w jednym wierzchołku może się zbiegać wiele krawędzi (jedna tabela może mieć wiele tabel podrzędnych).

12.2.3 Sygnatura nowych konstruktorów algebry kompozytów

SQL-owe konstruktory kompozytów zdefiniuję pomijając etap konstruktorów danych i korpusów. Będą one zapisane *implicite* w konstruktorach kompozytów. Zadbam o też o to, aby generowały błąd wszędzie tam — choć nie jedynie tam (!)¹²⁴ — gdzie błąd generują aplikacje SQL-owe. Ich nazwy będę tworzył zgodnie z zasadami przyjętymi w rozdziale 5.2.3 przyjmując, że wewnątrz kontekstu *Km[...]* znajduje się nazwa konstruktora danych *implicite* zdefiniowanego w definicji konstruktora kompozytów.

Teoriomnogościowo wiersze to po prostu rekordy, jednakże odpowiadające im kompozyty nie są rekordowymi, a więc postaci (*rek*, (*'R'*, *rko*)), ale wierszowymi, a więc postaci (*wie*, (*'Wq'*, *rko*)). Również wykonywane na nich operacje będą nieco inne niż rekordowe, choć często dość im bliskie.

Tworząc listę konstruktorów algebry kompozytów SQL-owych trzeba podjąć decyzję metodologiczną związaną z wyborem jednej z dwóch opcji:

1. każdej przyszłej konstrukcji językowej odpowiada odrębny konstruktor naszej algebry,

¹²³ Ponieważ zbiór identyfikatorów jest skończony, więc tym samym i każdy graf podrzędności jest skończony.

¹²⁴ W *Lingua-SQL* nie będzie sytuacji, o których autorzy SQL-owych podręczników piszą (rozdział 11.2), że „nie dostarczając (...) funkcjom prawidłowych wartości (...), nie należy oczekiwać rozsądnych wyników”. We wszystkich takich sytuacjach będzie generowany komunikat błędu.

2. każda przyszła konstrukcja językowa daje się opisać jako pewna kombinacja konstruktorów algebry.

Na przykład, operację wymiany danej stojącej w polu wiersza w tabeli można opisać jako jeden konstruktor przekształcający tabelę w tabelę, lub jako złożenie konstruktorów wymiany danej w wierszu i wiersza w tabeli.

Pierwsza opcja wydaje się bliższa tradycji SQL-owych podręczników, jednak prowadzi do dużych list konstrukcji „na każdą okazję” i przekłada się — jak sądzę — na gorsze rozumienie semantyki języka. Wybieram zatem drugą co powinno pozwolić na uzyskanie trzech efektów:

1. skrócenie i uproszczenie opisu języka,
2. skrócenie i uproszczenie listy reguł budowania metaprogramów (por. rozdział 8.4.3),
3. ograniczenie źródłowego kodu interpretera **Lingua-SQL** do podstawowych konstrukcji, a następnie zdefiniowanie pozostałych już na poziomie języka za pomocą mechanizmu procedur.

W poniższej sygnaturze pomijam konstruktory kompozytów prostych, które traktuję jako parametry modelu. Pozostałe konstruktory dzielę na kategorie odpowiadające kategoriom konstruktorów danych.

Dla zdefiniowania niektórych z niżej wymienionych konstruktorów odwołuję się do pojęcia transferu, które w rozdziale 12.4 rozszerzę na nowy zbiór kompozytów. Nową dziedzinę transferów oznaczam tak, jak poprzednią, a więc przez **Transfer** i ponownie zakładam, że obejmuje jedynie transfery transparentne względem błędów.

Konstruktory kompozytów wierszowych

Km[twórz-wi]	: Identyfikator x KompozytB	↦ KompozytB
Km[dołącz-do-wi]	: Identyfikator x KompozytB x KompozytB	↦ KompozytB
Km[usuń-z-wi]	: Identyfikator x KompozytB	↦ KompozytB
Km[weź-z-wi]	: Identyfikator x KompozytB	↦ KompozytB
Km[zmień-w-wi]	: Identyfikator x KompozytB x Transfer	↦ KompozytB

Wierszowe konstruktory kompozytów tabelowych

Km[twórz-tb-pustą]	: KompozytB	↦ KompozytB
Km[dołącz-wi-do-tb]	: KompozytB x KompozytB	↦ KompozytB
Km[usuń-wi-z-tb]	: Transfer x KompozytB	↦ KompozytB
Km[weź-wi-z-tb]	: Transfer x KompozytB	↦ KompozytB
Km[wyklucz-wi-z-tb]	: KompozytB x KompozytB	↦ KompozytB
Km[filtruj-wi-w-tb]	: Transfer x KompozytB	↦ KompozytB
Km[połącz-tb]	: KompozytB x KompozytB	↦ KompozytB
Km[przetnij-tb]	: KompozytB x KompozytB	↦ KompozytB

Kolumnowe konstruktory kompozytów tabelowych

$Km[do\acute{ł}o\acute{z}-ko-do-tb]$	: Identyfikator x KompozytB x KompozytB	$\mapsto$ KompozytB
$Km[usu\acute{n}-ko-z-tb]$	: Identyfikator x KompozytB	$\mapsto$ KompozytB
$Km[filtruj-ko-z-tb]$	: ParAkt x KompozytB	$\mapsto$ KompozytB
$Km[zmie\acute{n}-ko-w-tb]$	: Identyfikator x KompozytB x Transfer	$\mapsto$ KompozytB
$Km[we\acute{z}-ko-z-tb]$	: Identyfikator x KompozytB	$\mapsto$ KolumnaB

Tabelowy konstruktor tworzący tabelę pochodną

$tw\acute{o}rz-tb-poch : KompozytB \times KompozytB \times Identyfikator \times Transfer \mapsto$
KompozytB

Ostatni konstruktor nie powstaje z żadnego konstruktora danych, jego nazwa nie zawiera więc kontekstu $Km[\dots]$.

12.2.4 Konstruktory kompozytów prostych

Przyjmuję, że konstruktory kompozytów prostych w **Lingua-SQL** obejmują:

- wszystkie konstruktory kompozytów prostych z **Lingua-3**,
- konstruktory zeroargumentowe generujące nowe kompozyty proste,
- jakiś repertuar operacji i predykatów na takich kompozytach, których przykłady zostały podane w rozdziale 11.2.

Ten zbiór konstruktorów przyjmuję jako parametr modelu. Do tego dochodzi konstruktor, który każdemu korpusowi prostemu przypisuje kompozyt z daną pustą (odpowiadającą pustemu polu wiersza lub tabeli):

$pusty : KorpusB \mapsto KompozytB$

$pusty.kor =$

$kor : B\acute{ł}ad \rightarrow kor$

nie $kor : KorProsty \rightarrow \text{'oczekiwany-korpus-prosty'}$

prawda $\rightarrow (\Theta, kor)$

Ponieważ założyliśmy wcześniej, że Θ należy do klanu każdego korpusu, każdy (Θ, kor) jest poprawnym kompozytem.

12.2.5 Konstruktory kompozytów wierszowych

SQL-owe konstruktory wierszowe, choć zbliżone do konstruktorów rekordowych (rozdział 5.2.3), różnią się od nich dwiema cechami:

1. pozwalają na budowanie jedynie takich wierszy, które atrybutom przypisują dane proste,
2. przypisywaną atrybutowi daną może być Θ , co oznacza, że tworzymy wiersz z polem pustym, w które w przyszłości może być wstawiona dana o odpowiednim korpusie.

A oto przykłady definicji trzech spośród tych pięciu konstruktorów wierszowych wymienionych w rozdziale 12.2.3.

Dołącz atrybut do wiersza

$$\text{Km}[\text{dołącz-do-wi}] : \text{Identyfikator} \times \text{KompozytB} \times \text{KompozytB} \mapsto \text{KompozytB}$$

$$\text{Km}[\text{dołącz-do-wi}].(\text{ide}, \text{kom-p}, \text{kom-w}) = \quad (\text{p} - \text{prosty}, \text{w} - \text{wierszowy})$$

$$\text{kom-i} : \text{Błąd} \quad \rightarrow \text{kom-i} \quad \text{dla } i = \text{p}, \text{w}$$
niech

$$(\text{dan-p}, \text{kor-p}) = \text{kom-p}$$

$$(\text{dan-w}, \text{kor-w}) = \text{kom-w}$$

$$\text{nie kor-p} : \text{KorProsty} \quad \rightarrow \text{'oczekiwana-dana-prosta'}$$

$$\text{rodzaj.kor-w} \neq \text{'Wq'} \quad \rightarrow \text{'oczekiwany-wiersz'}$$

$$\text{dan-w.ide} = ! \quad \rightarrow \text{'atrybut-zajęty'}$$
niech

$$(\text{'Wq'}, \text{rko}) = \text{kor-w}$$

$$\text{nowy-kom} = (\text{dan-w}[\text{ide}/\text{dan-p}], (\text{'Wq'}, \text{rko}[\text{ide}/\text{kor-p}]))$$

$$\text{nadmiarowy.nowy-kom} \rightarrow \text{'przekroczenie-rozmiaru'}$$

$$\text{prawda} \quad \rightarrow \text{nowy-kom}$$

Przypominam (rozdział 5.2.2), że *rko* oznacza *rekord korpusów* korpusu wierszowego *kor-w*, który jest korpusem rekordowym. Dołożenie atrybutu do kompozytu wierszowego powiększa zarówno wiersz jak i odpowiednio jego korpus, dzięki czemu konstruktor zachowuje dobre ustrukturyzowanie kompozytu.

Zauważmy też, że nasz konstruktor na pozycji drugiego argumentu oczekuje kompozytu prostego, a na poziomie trzeciego — wiersza. W ten właśnie sposób realizowane jest ograniczenie konstruktorów SQL-owych do danych SQL-owych.

Weź daną z wiersza

$$\text{Km}[\text{weź-z-wi}] : \text{Identyfikator} \times \text{KompozytB} \mapsto \text{KompozytB}$$

$$\text{Km}[\text{weź-z-wi}].(\text{ide}, \text{kom}) =$$

$$\text{kom} : \text{Błąd} \quad \rightarrow \text{kom}$$
niech

$$(\text{dan}, \text{kor}) = \text{kom}$$

$$\text{rodzaj.kor} \neq \text{'Wq'} \quad \rightarrow \text{'oczekiwany-wiersz'}$$

$$\text{dan.ide} = ? \quad \rightarrow \text{'brak-takiego-atrybutu'}$$

$$\text{dan.ide} = \Theta \quad \rightarrow \text{'pole-puste'}$$
niech

$$(\text{'Wq'}, \text{rko}) = \text{kor}$$

$$\text{prawda} \quad \rightarrow (\text{dan.ide}, \text{rko.ide})$$

Zmień daną w wierszu warunkowo

$\text{Km}[\text{zmień-w-wi}] : \text{Identyfikator} \times \text{KompozytB} \times \text{Transfer} \times \text{Transfer} \mapsto \text{KompozytB}$

$\text{Km}[\text{zmień-w-wi}].(\text{ide}, \text{kom}, \text{tra}, \text{jar}) =$

$\text{kom} : \text{Błąd} \rightarrow \text{kom}$

$\text{rodzaj.kom} \neq \text{'Wq'} \rightarrow \text{'oczekiwany-wiersz'}$

niech

$(\text{wie}, \text{kor}) = \text{kom}$

$\text{wie.ide} = ? \rightarrow \text{'brak-takiego-atrybutu'}$

niech

$(\text{'Wq'}, \text{rko}) = \text{kor}$

$\text{tra.kom} : \text{Błąd} \rightarrow \text{tra.kom}$

$\text{jar.kom} : \text{Błąd} \rightarrow \text{jar.kom}$

niech

$(\text{dan}, \text{kor}) = \text{tra.kom}$

$(\text{dan-j}, \text{kor-j}) = \text{jar.kom}$

$\text{kor} \neq \text{rko.ide} \rightarrow \text{'korpusy-niezgodne'}$

$\text{kor-j} \neq (\text{'boolowska'}) \rightarrow \text{'oczekiwane-jarzmo'}$

niech

$\text{nowy-kom} =$

$\text{dan-j} = \text{pp} \rightarrow (\text{wie}[\text{ide/dan}], \text{kor})$

prawda $\rightarrow \text{kom}$

$\text{nadmiarowy.nowy-kom} \rightarrow \text{'przekroczenie-rozmiaru'}$

prawda $\rightarrow \text{nowy-kom}$

Nowa dana *dan*, która ma być przypisana w wierszu *wie* identyfikatorowi *ide*, powstaje w wyniku zastosowania transferu *tra* do kompozytu wierszowego *kom*. Podstawienie następuje jedynie wtedy, gdy kompozyt wierszowy spełnia jarzmo *jar*. Przed wstawieniem nowej danej następuje sprawdzenie, czy jej korpus jest zgodny z korpusem przypisanym w wierszu identyfikatorowi *ide*. Transfery SQL-owe zostaną opisane w rozdziale 12.4.

12.2.6 Wierszowe konstruktory kompozytów tabelowych

Konstruktory kompozytów tabelowych posłużą do zdefiniowania w przyszłości zarówno transformacji tabel, jak i tworzenia perspektyw oraz zapytań. Te konstruktory dzielę na dwie grupy: *konstruktory wierszowe* i *konstruktory kolumnowe*. Zanim je zdefiniuję wprowadzę pojęcie pomocnicze.

Powiem, że *korpus wierszowy kor jest zgodny z korpusem tabelowym ('Tq', wie, kor-w)*, jeżeli $\text{kor} = \text{kor-w}$.

Wprowadzę też funkcję pomocniczą, która pobiera jako argumenty dwa wiersze nad wspólnym zbiorem atrybutów:

wie-1 = [ide-1/dan-11,...,ide-n/dan-1n]

wie-2 = [ide-1/dan-21,...,ide-n/dan-2n]

i tworzy z nich nowy wiersz

uzupełnij.(wie-1, wie-2) = [ide-1/dan-31,...,ide-n/dan-3n]

powstający z drugiego przez wstawienie na miejsce każdej pary $\text{ide-}i/\Theta$ odpowiadającej jej pary w wierszu pierwszym, a więc $\text{ide-}i/\text{dan-}1i$. Tak więc dla $i = 1;n$

$\text{dan-}3i =$

$\text{dan-}2i \neq \Theta \rightarrow \text{dan-}2i$

$\text{dan-}2i = \Theta \rightarrow \text{dan-}1i$

Ta funkcja posłuży do zapisania zasady, że przy dodawaniu nowego wiersza do tabeli, jeżeli w tym wierszu jakieś pola nie zostały wypełnione, to należy je wypełnić danymi domyślnymi.

W poniższych definicjach konstruktorów wierszowych będę się odwoływał do operacji na krotkach zdefiniowanych w rozdziale 2.1.4.

Twórz tabelę pustą

$\text{Km}[\text{twórz-tb-pusta}] : \text{KompozytB} \mapsto \text{KompozytB}$

$\text{Km}[\text{twórz-tb-pusta}].\text{kom} =$

$\text{kom} : \text{Błąd} \rightarrow \text{kom}$

$\text{rodzaj.kom} \neq 'Wq' \rightarrow \text{'oczekiwany-wiersz'}$

niech

$(\text{wie}, \text{kor}) = \text{kom}$

prawda $\rightarrow ((), ('Tq', \text{wie}, \text{kor}))$

Tabela pusta powstaje w kompozycie wierszowego, którego wiersz staje się wierszem wartości domyślnych tabeli, a korpus wskazuje korpusy atrybutów.

Dołóż wiersz do tabeli

$\text{Km}[\text{dołóż-wi-do-tb}] : \text{KompozytB} \times \text{KompozytB} \mapsto \text{KompozytB}$

$\text{Km}[\text{dołóż-wi-do-tb}].(\text{kom-w}, \text{kom-t}) =$

$\text{kom-}i : \text{Błąd} \rightarrow \text{kom-}i \quad \text{dla } i = w, t$

$\text{rodzaj.kom-w} \neq 'Wq' \rightarrow \text{'oczekiwany-wiersz'}$

$\text{rodzaj.kom-t} \neq 'Tq' \rightarrow \text{'oczekiwana-tabela'}$

niech

$(\text{wie}, ('Wq', \text{kor-w})) = \text{kom-w}$

$(\text{tab}, \text{kor-t}) = \text{kom-t}$

$(\text{'Tq'}, \text{wie-d}, \text{kor-wt})) = \text{kor-t}$ (wt – wierszowy korpus tabeli)
 $\text{kor-w} \neq \text{kor-wt} \rightarrow \text{'niezgodne-korpusy'}$
niech
 $\text{wie-uz} = \text{uzupełnij.}(\text{wie-d}, \text{wie})$
 $\text{nowa-tab} = \text{tab} \odot (\text{wie-uz})$
 $\text{nowy-kom-t} = (\text{nowa-tab}, \text{kor-t})$
 $\text{są-powtórzenia.nowa-tab} \rightarrow \text{'wiersz-redundantny'}$
 $\text{nadmiarowy.nowy-kom-t} \rightarrow \text{'przekroczenie-rozmiaru'}$
prawda $\rightarrow \text{nowy-kom-t}$

Korpus dokładanego wiersza musi być zgodny z korpusem tabeli. Dodatkowo, jeżeli wartość jakiegoś atrybutu jest w dokładanym wierszu pusta, to w to miejsce jest podstawiana wartość tego atrybutu w wierszu wartości domyślnych (która też może być pusta). Oczywiście operacja dokładania wiersza nie zmienia korpusu tabeli.

Tabela, do której dokładamy wiersz może być pusta. Przy dokładaniu wiersza do tabeli dbamy też o to, aby nie wprowadzić do tabeli powtórzeń, przy czym chodzi o powtórzenie całych wierszy, a nie o powtórzenia w kolumnach. Wyeliminowanie powtórzeń w kolumnach (jeżeli okaże się oczekiwane) będzie opisane dopiero na poziomie denotacji, gdzie będą dostępne jarzma (rozdział 12.7)

Usuń wiersz z tabeli

$\text{Km}[\text{usuń-wi-z-tb}] : \text{Transfer} \times \text{KompozytB} \mapsto \text{KompozytB}$

$\text{Km}[\text{usuń-wi-z-tb}].(\text{tra}, \text{kom}) =$

$\text{kom} : \text{Błąd} \rightarrow \text{kom}$
 $\text{rodzaj.kom} \neq \text{'Tq'} \rightarrow \text{'oczekiwana-tabela'}$

niech

$(\text{tab}, (\text{'Tq'}, \text{wie-d}, \text{kor})) = \text{kom}$

$(\text{wie-1}, \dots, \text{wie-n}) = \text{tab}$

$n = 1 \rightarrow \text{'jedynego-wiersza-nie-można-usunąć'}$

$\text{tra.}(\text{wie-i}, \text{kor}) = (\text{pp}, (\text{'boolowska'})) \rightarrow (\text{dan}[\text{i/?}], \text{kor}) \quad \text{dla } i = 1;n$

prawda $\rightarrow \text{'brak-takiego-wiersza'}$

Ten konstruktor usuwa pierwszy wiersz tabeli, który spełnia transfer tra. Jeżeli nie ma takiego wiersza, to pojawia się komunikat błędu. W powyższej definicji $\text{dan}[\text{i/?}]$ oznacza (patrz rozdział 2.1.3) krotkę wierszy dan, z której usunięto i-ty element¹²⁵.

¹²⁵ Czytelnikom zaznajomionym ze standardem SQL znany jest fakt, że usuwanie wiersza z tabeli może być zablokowane przez więzy nadrzędności tabel, bądź też może powodować kaskadowe usuwanie wierszy z tabel podrzędnych. Tego rodzaju mechanizmy będą opisane w naszym modelu dopiero na

Weź wiersz z tabeli

$$\text{Km}[\text{weź-wi-z-tb}] : \text{Transfer} \times \text{KompozytB} \mapsto \text{KompozytB}$$

$$\text{Km}[\text{weź-wi-z-tb}].(\text{tra}, \text{kom}) =$$

$$\text{kom} : \text{Błąd}$$

$$\rightarrow \text{kom}$$

$$\text{rodzaj.kom} \neq \text{'Tq'}$$

$$\rightarrow \text{'oczekiwana-tabela'}$$
niech

$$(\text{tab}, (\text{'Tq'}, \text{wie-d}, \text{kor})) = \text{kom}$$

$$(\text{wie-1}, \dots, \text{wie-n}) = \text{tab}$$

$$\text{tra}(\text{wie-i}, \text{kor}) = (\text{pp}, (\text{'boolowska'})) \rightarrow (\text{wie-i}, \text{kor}) \quad \text{dla } i = 1;n$$
prawda

$$\rightarrow \text{'brak-takiego-wiersza'}$$

Dla każdego wiersza tabeli jest tworzony kompozyt wierszowy składający się z tego wiersza oraz wierszowego korpusu tabeli. Następnie pobierany jest pierwszy taki kompozyt, który spełnia transfer tra. Jeżeli nie ma takiego kompozytu, to pojawia się komunikat błędu. Jednakże, gdyby w trakcie przeszukiwania tabeli któreś tra.(wie-i, kor) okazało się błędem, to przeszukiwanie będzie dalej. Zauważmy też, że skoro ('Tq', wie-d, kor) jest korpusem tabelowym, to kor musi być korpusem wierszowym.

Oczywiście, jeżeli tra nie będzie jarzmem wierszowym, to pojawi się komunikat 'brak-takiego-wiersza'. Występujący w tej definicji transfer tra będę nazywał *transferem selekcji*.

Wyklucz wiersze z tabeli

$$\text{Km}[\text{wyklucz-wi-z-tb}] : \text{KompozytB} \times \text{KompozytB} \mapsto \text{KompozytB}$$

$$\text{Km}[\text{wyklucz-wi-z-tb}].(\text{kom-1}, \text{kom-2}) =$$

$$\text{kom-i} : \text{Błąd}$$

$$\rightarrow \text{kom-i}$$

$$\text{dla } i = 1,2$$

$$\text{rodzaj.kom-i} \neq \text{'Tq'}$$

$$\rightarrow \text{'oczekiwana-tabela'}$$

$$\text{dla } i = 1,2$$
niech

$$(\text{tab-i}, \text{kor-i}) = \text{kom-i} \quad \text{dla } i = 1,2$$

$$\text{kor-1} \neq \text{kor-2}$$

$$\rightarrow \text{'niezgodne-korpusy'}$$
niech

$$\text{nowa-tab} = \text{różnica}(\text{tab-1}, \text{tab-2})$$

$$(\text{patrz. rozdział 2.1.4})$$

$$\text{nowy-kom} = (\text{nowa-tab}, \text{kor-1})$$
prawda

$$\rightarrow \text{nowy-kom}$$

Ten konstruktor usuwa z pierwszej tabeli wszystkie wiersze należące do drugiej tabeli. W wyniku tej operacji może powstać tabela pusta.

Kolejny konstruktor generuje tabelę obejmującą wszystkie wiersze zadanej tabeli spełniające zadany transfer.

Filtruj wiersze w tabeli

$Km[\text{filtruj-wi-w-tb}] : \text{Transfer} \times \text{KompozytB} \mapsto \text{KompozytB}$

$Km[\text{filtruj-wi-w-tb}].(\text{tra}, \text{kom}) =$

$\text{kom} : \text{Błąd} \rightarrow \text{kom}$

$\text{rodzaj.kom} \neq \text{'Tq'} \rightarrow \text{'oczekiwana-tabela'}$

niech

$((\text{wie-1}, \dots, \text{wie-n}), (\text{'Tq'}, \text{kor})) = \text{kom}$

$\text{po-krotka-kom-wie} = ((\text{wie-1}, \text{kor}), \dots, (\text{wie-n}, \text{kor}))$ (po – początkowa)

$\text{ko-krotka-kom-wie} = \text{filtruj.tra.po-krotka-kom-wie}$ (ko – końcowa)

$\text{ko-krotka-kom-wie} = () \rightarrow \text{'żaden-wiersz-nie-spełnia-warunku'}$

niech

$((\text{wie-ko-1}, \text{kor}), \dots, (\text{wie-ko-m}, \text{kor})) = \text{ko-krotka-kom-wie}$

prawda $\rightarrow ((\text{wie-ko-1}, \dots, \text{wie-ko-m}), (\text{'Tq'}, \text{kor-w}))$

W procesie tworzenia nowej tabeli wpierw jest tworzona początkowa krotka kompozytów wierszowych odpowiadających wszystkim wierszom tabeli. Ten zabieg jest konieczny, gdyż transfery są zdefiniowane na kompozytach, a nie na wierszach. Następnie z krotki początkowej powstaje krotka końcowa przez zastosowanie operatora filtruj (rozdział 2.1.4) korzystającego z zadanego transferu tra. W ten sposób powstaje końcowa krotka kompozytów wierszowych. Z tej krotki pobierane są wiersze do nowej tabeli. Oczywiście operacja wyboru wierszy nie zmienia korpusu tabeli.

Połącz dwie tabele

$Km[\text{połącz-tb}] : \text{KompozytB} \times \text{KompozytB} \mapsto \text{KompozytB}$

$Km[\text{połącz-tb}].(\text{kom-1}, \text{kom-2}) =$

$\text{kom-i} : \text{Błąd} \rightarrow \text{kom-i}$ dla $i = 1, 2$

$\text{rodzaj.kom-i} \neq \text{'Tq'} \rightarrow \text{'oczekiwana-tabela'}$ dla $i = 1, 2$

niech

$(\text{tab-i}, \text{kor-i}) = \text{kom-i}$ dla $i = 1, 2$

$\text{kor-1} \neq \text{kor-2} \rightarrow \text{'niezgodne-korpusy'}$

niech

$\text{nowa-tab} = \text{łącz-bez-powtórzeń}(\text{tab-1}, \text{tab-2})$ (rozdział 2.1.4)

$\text{nowy-kom} = (\text{nowa-tab}, \text{kor-1})$

nadmiarowy.nowy-kom $\rightarrow$ 'przekroczenie-rozmiaru'
prawda $\rightarrow$ nowy-kom

Połączenie dwóch tabel powoduje dołączenie do pierwszej tabeli tych wszystkich wierszy drugiej tabeli, które nie prowadzą do powtórzeń. Łączone tabele muszą mieć identyczne korpusy.

Przetnij dwie tabele

$Km[przetnij-tb] : \text{KompozytB} \times \text{KompozytB} \mapsto \text{KompozytB}$

$Km[przetnij-tb].(kom-1, kom-2) =$

$kom-i : \text{Błąd} \rightarrow kom-i$ dla $i = 1, 2$

$rodzaj.kom-i \neq 'Tq' \rightarrow \text{'oczekiwana-tabela'}$ dla $i = 1, 2$

niech

$(tab-i, kor-i) = kom-i$ dla $i = 1, 2$

$kor-1 \neq kor-2 \rightarrow \text{'niezgodne-korpusy'}$

niech

$nowa-tab = \text{część-wspólna}(tab-1, tab-2)$ (rozdział 2.1.4)

$nowy-kom = (nowa-tab, kor-1)$

prawda $\rightarrow$ nowy-kom

W wyniku działania tego konstruktora otrzymujemy tabelę zawierającą jedynie te wiersze, które były wspólne dla obu tabel wyjściowych. Przecinane tabele muszą mieć identyczne korpusy.

12.2.7 Kolumnowe konstruktory kompozytów tabelowych

Kolumną nazwiemy każdą niepustą krotkę kompozytów prostych. Zakładam też, że dziedzina kolumn obejmuje błędy, choć same kolumny błędów nie zawierają. Tak więc:

$kol : \text{KolumnaB} = \text{KomProsty}^{c+} \mid \text{Błąd}$

Z pojęciem kolumny są związane cztery niżej określone konstruktory tabel. Pierwsze trzy są związane implícite, gdyż żaden z nich ani nie przyjmuje kolumny jako swojego argumentu, ani też nie zwraca jej jako wyniku. Czwarty zwraca kolumnę jako wynik, ale ma charakter jedynie pomocniczy i nie będzie miał swojego odpowiednika na poziomie składni języka.

Kolumnowe konstruktory tabelowe są definiowane według jednolitej zasady sprowadzającej się do trzech kroków:

1. rozłożenie kompozytu tabelowego na ciąg kompozytów wierszowych,
2. modyfikacja każdego kompozytu wierszowego odpowiednim konstruktorem,
3. złożenie wynikowych kompozytów wierszowych w nowy kompozyt tabelowy (konstruktory *dołącz*, *usuń*, *zmień*) lub kolumnę (konstruktor *weź*).

Dołóż kolumnę do tabeli

$$\text{Km}[\text{dołóż-ko-do-tb}] : \text{Identyfikator} \times \text{KompozytB} \times \text{KompozytB} \mapsto \text{KompozytB}$$

$$\text{Km}[\text{dołóż-ko-do-tb}].(\text{ide}, \text{kom-p}, \text{kom-t}) = \quad (\text{p} - \text{prosty}, \text{t} - \text{tabelowy})$$

$$\text{kom-i} : \text{Błąd} \quad \rightarrow \text{kom-i} \quad \text{dla } i = \text{p}, \text{t}$$

$$\text{rodzaj.kom-t} \neq \text{'Tq'} \quad \rightarrow \text{'oczekiwana-tabela'}$$
niech

$$(\text{tab}, (\text{'Tq'}, \text{wie-d}, \text{kor-w})) = \text{kom-t}$$

$$(\text{dan-p}, \text{kor-p}) = \text{kom-p}$$

$$(\text{wie-1}, \dots, \text{wie-n}) = \text{tab}$$

$$\text{kom-j} = (\text{wie-j}, \text{kor-w}) \quad \text{dla } j = 1; n \quad (1)$$

$$\text{kom-d} = (\text{wie-d}, \text{kor-w})$$

$$\text{nowy-kom-j} = \text{dołóż-do-wi}(\text{ide}, \text{kom-p}, \text{kom-j}) \quad \text{dla } j = 1; n \quad (2)$$

$$\text{nowy-kom-d} = \text{dołóż-do-wi}(\text{ide}, \text{kom-p}, \text{kom-d})$$

$$\text{nowy-kom-j} : \text{Błąd} \quad \rightarrow \text{nowy-kom-j} \quad \text{dla } j = 1; n$$

$$\text{nowy-kom-d} : \text{Błąd} \quad \rightarrow \text{nowy-kom-d}$$
niech

$$(\text{nowy-wie-j}, \text{nowy-kor-w}) = \text{nowy-kom-j} \quad \text{dla } j = 1; n$$

$$(\text{nowy-wie-d}, \text{nowy-kor-w}) = \text{nowy-kom-d}$$

$$\text{nowa-tab} = (\text{nowy-wie-1}, \dots, \text{nowy-wie-n}) \quad (3)$$

$$\text{nowy-kom-t} = (\text{nowa-tab}, (\text{'Tq'}, \text{nowy-wie-d}, \text{nowy-kor-w})) \quad (4)$$

$$\text{nadmiarowy.nowy-kom-t} \rightarrow \text{'przekroczenie-zakresu'}$$

$$\text{prawda} \rightarrow \text{nowy-kom-t}$$

Zdefiniowany konstruktor dodaje do tabeli nową kolumnę wypełnioną we wszystkich wierszach tabeli, oraz w wierszu danych domyślnych, tą samą daną pochodzącą z kompozytu prostego kom-p. Ta konstrukcja dokonuje się w czterech krokach:

1. Po przeprowadzeniu wszystkich koniecznych sprawdzeń z kompozytu tablicowego kom-t tworzy się rodzinę kompozytów wierszowych kom-j o jednakowych korpusach kor-w pochodzących z korpusu tablicy. Do tego dochodzi kompozyt kom-d wiersza danych domyślnych.
2. Każdy z tak powstałych kompozytów wierszowych jest rozszerzany o nowy atrybut za pomocą konstruktora wierszy dołóż-do-wi (rozdział 12.2.5). Ten konstruktor jest odpowiedzialny również za sprawdzenie, czy kom-p jest kompozytem prostym, a także czy identyfikator ide nie występuje już w zbiorze atrybutów tabeli. Wszystkie powstałe w ten sposób kompozyty mają wspólny korpus nowy-kor-w.
3. Tak zbudowane nowe wiersze nowy-wie-j tworzą nową tabelę nowa-tab.
4. Nowy kompozyt tabelowy składa się z nowej tabeli i nowego korpusu tabelowego ('Tq', nowy-wie-d, nowy-rko-w).

Oczywiście powyższy algorytm nie musi być podstawą do napisania procedury implementującej nasz konstruktor. Służy on jedynie do zdefiniowania efektu działania konstruktora.

Usuń kolumnę z tabeli

$Km[usuń-ko-z-tb] : Identyfikator \times KompozytB \mapsto KompozytB$

$Km[usuń-ko-z-tb].(ide, kom-t) =$

$kom-t : Błąd \rightarrow kom-t$

$rodzaj.kom-t \neq 'Tq' \rightarrow 'oczekiwana-tabela'$

niech

$(tab, ('Tq, wie-d, kor-w) = kom-t$

$('Wq', rko) = kor-w$

$(\exists ide) dom.rko = \{ide\} \rightarrow 'jedynej-kolumny-nie-można-usunąć'$

niech

$(wie-1, \dots, wie-n) = tab$

$kom-j = (wie-j, kor-w) \quad \text{dla } j = 1;n$

$kom-d = (wie-d, kor-w)$

$nowy-kom-j = Km[usuń-z-wi].(ide, kom-j) \quad \text{dla } j = 1;n$

$nowy-kom-d = Km[usuń-z-wi].(ide, kom-d)$

$nowy-kom-j : Błąd \rightarrow nowy-kom-j \quad \text{dla } j = 1;n$

$nowy-kom-d : Błąd \rightarrow nowy-kom-d$

niech

$(nowy-wie-j, nowy-kor-w) = nowy-kom-j \quad \text{dla } j = 1;n$

$(nowy-wie-d, nowy-kor-w) = nowy-kom-d$

$nowa-tab = (nowy-wie-1, \dots, nowy-wie-n)$

$nowy-kom = (nowa-tab, ('Tq', nowy-wie-d, nowy-kor-w))$

prawda $\rightarrow nowy-kom$

Ten konstruktor jest zdefiniowany analogicznie do poprzedniego, z tym że odwołuje się do konstruktora wierszy $Km[usuń-z-wi]$ (rozdział 12.2.3), który m.in. sprawdza, czy *ide* jest atrybutem zadanej tabeli. Ze względów oczywistych pominięte zostało sprawdzenie, czy nie pojawiło się przekroczenie zakresu. Pojawiło się natomiast sprawdzenie, czy usuwana kolumna nie jest jedyną kolumną tabeli.

Filtruj wskazane kolumny z tabeli (usuń niewskazane)

$Km[filtruj-ko-z-tb] : ParAkt \times KompozytB \mapsto KompozytB$

$Km[filtruj-ko-z-tb].(pak, kom) =$

$pak = () \rightarrow 'wskaż-jakiś-atrybut'$

kom : Błąd $\rightarrow$ kom
 rodzaj.kom $\neq$ 'Tq' $\rightarrow$ 'oczekiwana-tabela'
niech
 (ide-1,...,ide-n) = pak
 (tab, ('Tq', wie, ('Wq', rko))) = kom
 rko.ide-i = ? $\rightarrow$ 'brak-atrybutu-ide-i' dla i = 1;n
niech
 tab-fi = tab ogr {ide-1,...,ide-n}
 wie-fi = wie ogr {ide-1,...,ide-n}
 rko-fi = rko ogr {ide-1,...,ide-n}
prawda $\rightarrow$ (tab-fi, ('Tq', wie-fi, ('Wq', rko-fi)))

W definicji tego konstruktora wykorzystałem dziedzinę parametrów aktualnych ParAkt wprowadzoną dla procedur w rozdziale 7.1.4, choć w tym przypadku gra ona inną rolę. Konstruktor usuwa z tabeli wszystkie kolumny poza tymi, których atrybuty występują na liście parametrów. Operator ogr został zdefiniowany w rozdziale 2.1.3. Zauważmy, że obecność powtórzeń na liście parametrów nie wpływa na wynik działania konstruktora.

Zmień kolumnę w tabeli warunkowo

Km[zmień-ko-w-tb] : Identyfikator x KompozytB x Transfer x Transfer
 $\mapsto$ KompozytB

Km[zmień-ko-w-tb].(ide, kom, tra) =
 kom : Błąd $\rightarrow$ kom
 rodzaj.kom $\neq$ 'Tq' $\rightarrow$ 'oczekiwana-tabela'
niech
 (tab, ('Tq, wie, kor) = kom
 (wie-1,...,wie-n) = tab
 kom-j = (wie-j, kor) dla j = 1;n
 nowy-kom-j = Km[zmień-w-wi].(ide, kom-j, tra, jar) dla j = 1;n
 nowy-kom-j : Błąd $\rightarrow$ nowy-kom-j dla j = 1;n
niech
 (nowy-wie-j, kor) = nowy-kom-j dla j = 1;n
 nowa-tab = (nowy-wie-1,...,nowy-wie-n)
 nowy-kom = (nowa-tab, ('Tq', wie, kor))
prawda $\rightarrow$ nowy-kom

Ten konstruktor stosuje konstruktor wierszy $Km[zmień-w-wi]$ do każdego wiersza tabeli. Zastosowanie tego konstruktora nie zmienia korpusu tabeli, może jednak spowodować wygenerowanie błędu niezgodności korpusów przez konstruktor wierszowy. Szczególnym przypadkiem użycia tego konstruktora może być denotacja instrukcji:

```
UPDATE Pracownicy
SET Pensja = Pensja * 1,1
WHERE Stanowisko = 'sprzedawca'
```

Ostatni konstruktor z tej grupy dokonuje selekcji kolumny z tabeli. Takiego konstruktora nie ma chyba w standardzie SQL, wprowadzam go jednak, gdyż będzie potrzebny przy określaniu jarzm dla tabel. W **Lingua-SQL** jego denotacyjny odpowiednik nie należy do sygnatury algebry denotacji, nie będzie więc dostępny z poziomu składni.

Weź kolumnę z tabeli

$Km[weź-ko-z-tb] : Identyfikator \times KompozytB \mapsto KolumnaB$

$Km[weź-ko-z-tb].(ide, kom-t) =$

$kom-t : Błąd \rightarrow kom-t$

$rodzaj.kom \neq 'Tq' \rightarrow 'oczekiwana-tabela'$

niech

$(tab, ('Tq', wie-d, kor-w)) = kom-t$

$(wie-1, \dots, wie-n) = tab$

$kom-j = (wie-j, kor-w) \quad \text{dla } j = 1;n$

$wybrany-kom-j = Km[weź-z-wi].(ide, kom-j) \quad \text{dla } j = 1;n$

$wybrany-kom-j : Błąd \rightarrow nowy-kom-j \quad \text{dla } j = 1;n$

prawda $\rightarrow (wybrany-kom-1, \dots, wybrany-kom-n)$

Ten konstruktor tworzy krotkę kompozytów prostych odpowiadających atrybutowi *ide* w każdym z wierszy tabeli poza wierszem wartości domyślnych. Zatem pobrana kolumna nie zawiera kompozytu domyślnego.

12.2.8 Referencyjny konstruktor kompozytów tabelowych

Ten konstruktor służy do łączenia informacji z dwóch tabel dla zbudowania trzeciej. W typowych zastosowaniach tabele źródłowe będą połączone relacją podrzędności, jednak w definicji konstruktora nie będę korzystał z tego założenia, gdyż grafy podrzędności będą dostępne dopiero na poziomie konstruktorów denotacji¹²⁶. Rozpocznę od zdefiniowania konstruktora pomocniczego, który przyjmuje trzy argumenty:

1. kompozyt niosący wiersz *kom-w*.

¹²⁶ Alternatywą dla takiego rozwiązania mogłoby być dodanie do algebry kompozytów trzeciego nośnika, którym byłaby dziedzina grafów podrzędności. Nie wybrałem jednak tego rozwiązania, gdyż nie chciałem zmieniać sygnatury algebry kompozytów. Prawdę mówiąc nie potrafię w tej chwili powiedzieć, które z tych rozwiązań jest lepsze.

2. indyktor nadrzędności *ide*,
3. kompozyt niosący tabelę *kom-t*

a oddaje kompozyt pierwszego wiersza tabeli *kom-t*, na który wskazuje wiersz niesiony przez *kom-w* przez identyfikację wspólną w obu wierszach wartości przypisanej identyfikatorowi nadrzędności.

Wskazany wiersz tabeli niesionej przez *kom-t* nazwę *wierszem nadrzędnym* dla wiersza *kom-w*. Taką relację pomiędzy wierszami ilustruje Rys. 12.2-1. Jeżeli wiersz *kom-w* pochodzi z tabeli podrzędnej wobec *kom-t*, to wiersz nadrzędny zawsze istnieje i jest wyznaczony jednoznacznie. W przeciwnym przypadku może nie istnieć, ale może też być ich więcej niż jeden.

Tabelowy konstruktor wskazujący wiersz nadrzędny

wskaz-wi-nad : KompozytB x Identyfikator x KompozytB $\mapsto$ KompozytB

wskaz-wi-nad.(*kom-w*, *ide*, *kom-t*) =

kom-i : Błąd $\rightarrow$ *kom-i* dla $i = w, t$

rodzaj.kom-w $\neq$ 'Wq' $\rightarrow$ 'oczekiwany-wiersz'

rodzaj.kom-t $\neq$ 'Tq' $\rightarrow$ 'oczekiwana-tabela'

niech

(*wie*, ('Wq', *rko-w*)) = *kom-w*

(*tab*, ('Tq', *wie-d*, ('Wq', *rko-t*))) = *kom-t*

rko-i.ide = ? $\rightarrow$ 'atrybut-nieznany' dla $i = w, t$

niech

(*wie-1*, ..., *wie-n*) = *tab*

wie-i.ide = *wie.ide* $\rightarrow$ (*wie-i*, ('Wq', *rko-t*))

prawda $\rightarrow$ 'brak-takiego-wiersza'

Po koniecznych sprawdzeniach tabela źródłowa *kom-t* jest przeglądana wiersz po wierszu w poszukiwaniu wiersza, który pod atrybutem *ide* niesie tę samą daną co wiersz źródłowy *kom-w*. Pierwszy taki wiersz, jeżeli zostanie znaleziony, staje się wynikiem konstruktora. W przeciwnym przypadku ukazuje się komunikat błędu.

Teraz możemy zdefiniować nasz konstruktor docelowy, który przyjmuje cztery argumenty:

1. tabelę *kom-p* „grającą rolę” podrzędnej,
2. tabelę *kom-n* „grającą rolę” nadrzędnej,
3. wspólny dla obu tabel indyktor *ide*,
4. transfer wierszowy *tra*.

i tworzy z nich tabelę tych wierszy tabeli *kom-p*, które wskazują na wiersze tabeli *kom-n* spełniające transfer *tra*. Oczywiście aby taki konstruktor wygenerował tabelę, *tra* musi być narzędziem, choć rzecz jasna nie jest to warunek wystarczający. Tabelę powstającą w wyniku działania tego konstruktora nazwę *tabelą pochodną* dla pary tabel źródłowych.

Tabelowy konstruktor tworzący tabelę pochodną referencyjnie

$\text{twórz-tb-ref} : \text{KompozytB} \times \text{KompozytB} \times \text{Identyfikator} \times \text{Transfer} \mapsto \text{KompozytB}$

Ten konstruktor zdefiniuję przez indukcję względem liczby wierszy w tabeli podrzędnej. Rozpocynam więc od tabeli zawierającej tylko jeden wiersz. Dla tego przypadku tworzę odrębny konstruktor:

$\text{twórz-tb-ref-1w}(\text{kom-p}, \text{kom-n}, \text{ide}, \text{tra}) =$

$\text{kom-i Błąd} \rightarrow \text{kom-i} \quad \text{dla } i = p, n$

$\text{rodzaj.kom-i} \neq \text{'Tq'} \rightarrow \text{'oczekiwana-tabela'}$

niech

$((\text{wie}), (\text{'Tq'}, \text{wie-d}, \text{kor-w})) = \text{kom-p}$

$\text{kom-w} = (\text{wie}, \text{kor-w}) \quad (\text{tworzę kompozyt z wiersza tabeli})$

$\text{kom-wn} = \text{wskaż-wi-nad}(\text{kom-w}, \text{ide}, \text{kom-n})$

$\text{kom-wn} : \text{Błąd} \rightarrow \text{kom-wn}$

$\text{tra.kom-wn} = (\text{pp}, (\text{'boolowska'})) \rightarrow \text{kom-p}$

$\text{tra.kom-wn} : \text{Błąd} \rightarrow \text{tra.kom-wn}$

prawda $\rightarrow ((), (\text{'Tq'}, \text{wie-d}, \text{kor-w}))$

Zdefiniowany wcześniej konstruktor *wskaż-wi-nad* wskazuje kompozyt *kom-wn* niosący wiersz nadrzędny dla kompozytu *kom-w* niosącego jedyny wiersz tabeli *kom-p*. Jeżeli wiersz nadrzędny *kom-wn* spełnia transfer *tra*, to tabela wyjściowa *kom-p* staje się wynikiem konstruktora. W przeciwnym przypadku:

- jeżeli zastosowanie transferu prowadzi do błędu, to ten błąd jest sygnalizowany na zewnątrz,
- a w przeciwnym przypadku, tj. jeżeli transfer generuje fałsz, wynikiem konstruktora jest tabela pusta z korpusem tabeli wyjściowej *kom-p*.

Drugi krok indukcyjny jest następujący:

$\text{twórz-tb-ref}(\text{kom-p}, \text{kom-n}, \text{ide}, \text{tra}) =$

$\text{kom-i Błąd} \rightarrow \text{kom-i} \quad \text{dla } i = p, n$

$\text{rodzaj.kom-i} \neq \text{'Tq'} \rightarrow \text{'oczekiwana-tabela'}$

niech

$(\text{tab}, \text{kor-t}) = \text{kom-p}$

$\text{tab} = () \rightarrow \text{'pusta-tabela-podrzędna'}$

niech

$((\text{wie-1}, \dots, \text{wie-k}), \text{kor-t}) = \text{kom-p}$

kom-p-1	= ((wie-1), kor-t))	
kom-wn-1	= twórz-tb-ref-1w.(kom-p-1, ide, kom-n)	
kom-wn-1 : Błąd	→ kom-wn-1	
k = 1	→ kom-wn-1	
niech		
kom-res	= ((wie-2,...,wie-k), kor-t)	(res – residuum)
kom-wn-res	= twórz-tb-poch.(kom-res, ide, kom-n)	
kom-wn-res : Błąd	→ kom-wn-res	
prawda	→ Km[połącz-tb].(kom-wn-1, kom-wn-ind)	

Po koniecznych sprawdzeniach jest tworzona tabela wynikowa kom-wn-1 dla tabeli kom-p-1 powstającej z kom-p przez pozostawienie w niej tylko pierwszego wiersza.

Jeżeli tabela kom-p miała tylko jeden wiersz, to na tym wyliczenie się kończy. W przeciwnym przypadku rekurencyjnie aplikujemy definiowany konstruktor do residuum tabeli kom-p. Otrzymaną w ten sposób tabelę doklejamy do tablicy otrzymanej z pierwszego wiersza za pomocą konstruktora łączenia tabel.

Zauważmy, że ten konstruktor jest zdefiniowany dla dowolnej pary tabel wyjściowych, tj. niekoniecznie połączonych relacją nadrzędności.

12.3 Korpusy

W rozdziale 12.2 zapowiedziałem, że konstruktory kompozytów SQL-owych zdefiniuję bezpośrednio, a nie przez odwoływanie się do konstruktorów danych i korpusów. Ponieważ jednak niektóre konstruktory korpusów stanowią podstawę do zdefiniowania konstruktorów typów (rozdział 12.5) oraz definicji stałych typologicznych (rozdział 12.7.4), podaję poniżej listę tych konstruktorów, które będą mi do tego potrzebne. Zwracam uwagę, że nie są to wszystkie konstruktory korpusów zawarte w definicjach konstruktorów kompozytów, ale jedynie te, które zostaną wykorzystane przez konstruktory typów. Ich definicje pomijam bo są one implicite zawarte w definicjach konstruktorów kompozytów.

Konstruktory korpusów wierszowych

Kr[twórz-wi]	: Identyfikator x KorpusB	↦ KorpusB
Kr[dołącz-wi]	: Identyfikator x KorpusB x KorpusB	↦ KorpusB

Konstruktory korpusów tabelowych

Kr[twórz-tb-pustą]	: KompozytB	↦ KorpusB
Kr[dołącz-wi-tb]	: Identyfikator x KorpusB x KorpusB	↦ KorpusB
Kr[weź-wi-tb]	: KorpusB	↦ KorpusB

12.4 Transfery

O typach danych — tak jak rozumiemy je w niniejszej książce — mówi się w podręcznikach SQL-owych jedynie w kontekście danych prostych, a i w tym przypadku w sposób dalece niekompletny i niejasny. Typów tabel można się jakoś doszukać w deklaracjach tabel, a typy wierszy, kolumn i baz danych trzeba sobie dopowiedzieć. Na dodatek w deklaracjach tabel są wymieszane treści odpowiadające korpusom z treściami odpowiadającymi jarzmom (rozdział 11.3). Jedne i drugie w literaturze SQL-owej są nazywane *więzami spójności*.

Niestety spośród podręczników SQL, które udało mi się przewertować (ich listę podaję w preambule do rozdziału 11), nie znalazłem takiego, który podawałby kompletny opis więzów spójności. Choć wszystkie te opisy mają jakąś część wspólną, to poza nią praktycznie w każdym podręczniku można znaleźć co innego. W tej sytuacji postanowiłem zbudować na tyle uniwersalny model SQL-owych typów w sensie zdefiniowanym w rozdziale 5.2.5, aby dały się w nim zawrzeć, jeżeli nie wszystkie SQL-owe mechanizmy, to przynajmniej dostatecznie duża ich część. W rezultacie powstał model wykraczający poza standardowy SQL.

Ponieważ w modelu dla **Lingua-SQL** nie występują kompozyty bazodanowe, nie będzie też bazodanowych transferów. Właściwości baz danych będą opisywane przez:

- jarzma występujących w nich tabel oraz,
- grafy podrzędności, które pojawiają się jednak dopiero na poziomie denotacji instrukcji.

Przyjmuję, że **Lingua-SQL** są dostępne wszystkie zdefiniowane do tej pory konstruktory transferów, a w szczególności konstruktory boolowskie. Nowe transfery obejmą transfery związane z nowymi danymi prostymi, które potraktuję jako parametr modelu, oraz niżej zdefiniowane transfery wierszowe i tabelowe.

12.4.1 Transfery wierszowe

W tej grupie mamy tylko jeden nowy konstruktor. Jest on analogiczny do konstruktora selekcji danej z rekordu:

$\text{Kt}[\text{weż-z-wi}] : \text{Identyfikator} \mapsto \text{Transfer}$

$\text{Kt}[\text{weż-z-wi}].\text{ide.kom} = \text{Km}[\text{weż-z-wi}].(\text{ide}, \text{kom})$

Odwołując się to tego konstruktora można zdefiniować konstruktor jarzma sprawdzającego, czy zadany atrybut wiersza niesie zadaną wartość

$\text{niesie-pod-atr} : \text{Transfer} \times \text{Identyfikator} \mapsto \text{Transfer}$

$\text{niesie-pod-atr}(\text{tra}, \text{ide}).\text{kom} =$

$\text{Km}[\text{równe}].(\text{tra.kom}, \text{Km}[\text{weż-z-wi}].(\text{ide.kom}))$

Zauważmy, że skoro kom musi być kompozytem wierszowym, to tra może być albo transferem o wartości stałej będącej kompozytem prostym, albo transferem zbudowanym za pomocą $\text{Kt}[\text{weż-z-wi}]$. W tym pierwszym przypadku, jeżeli zachodzi na przykład równość:

$\text{tra.kom} = (\text{'zarząd'}, (\text{'słowo'}))$

to wtedy

$\text{niesie-pod-atr}(\text{tra}, \text{'dział'})$

będzie denotacją jarzma wierszowego

$\text{row.dział} = \text{'zarząd'}$

Pozostałe konstruktory, a w tym boolowskie, pozwalają tworzyć złożone transfery wierszowe w ten sam sposób jak dla rekordów. Przykładem takiego transferu, a w rzeczywistości jarzma, może być:

```
row.dział = 'zarząd' or
row.pensja + row.premia ≤ 7000
```

wyrażające ograniczenie na wynagrodzenia pracowników nie będących członkami zarządu. Takie jarzmo może być użyte zarówno przy definiowaniu typu tabelowego, jak i przy budowaniu zapytania. Przypominam, że **row** jest w tym przypadku słowem kluczowym, a nie zmienną.

12.4.2 Transfery tabelowe

Transfery tabelowe dzielą się one na dwie klasy. Do pierwszej należą *tabelowe jarzma kwantyfikatorskie* opisujące własności tabeli przez wskazanie jarzma wierszowego, które ma być spełniane przez wszystkie wiersze tabeli. Do drugiej grupy należą jarzma kolumnowe.

W pierwszym przypadku mamy tu sytuację analogiczną jak przy tworzeniu jarzma dla listy za pomocą konstruktora *wszystkie-na-li*. Nazwa tego konstruktora nie ma formy *Kt[ope]*, gdyż nie odwołuje się do żadnego konstruktora algebry danych.

Tabelowe jarzmo kwantyfikatorskie

wszystkie-w-tb : Transfer $\mapsto$ Transfer

wszystkie-w-tb.tra.kom =

kom : Błąd	$\rightarrow$ kom
rodzaj.kom $\neq$ 'Tq'	$\rightarrow$ 'oczekiwana-tabela'
niech	
((wie-1,...,wie-n), ('Tq', wie-d, kor))	= kom
kom-i	= (wie-i, kor) dla i = 1;n
tra.kom-i : Błąd	$\rightarrow$ tra.kom-i dla i = 1;n
rodzaj.(tra.kom-i) $\neq$ ('boolowska')	$\rightarrow$ 'oczekiwane-jarzmo'
($\forall$ kom-i) tra.kom-i = (pp, ('boolowska'))	$\rightarrow$ (pp, ('boolowska'))
prawda	$\rightarrow$ (ff, ('boolowska'))

Zauważmy, że tranzyt *tra* nie musi być spełniony przez wiersz danych domyślnych *wie-d*. Ta decyzja została podyktowana faktem, że niektóre pola wiersza *wie-d* mogą być puste.

Co do swojej natury tabelowe jarzma kwantyfikatorskie wyrażają explicite własności wierszy tabeli, ale też, dzięki użyciu kwantyfikatora ogólnego, wyrażają implicite pewne własności kolumn. Są to te wszystkie własności, które dają się wyrazić przez wskazania jarzma, które ma być spełnione przez wszystkie elementy kolumny. Przy pomocy tej techniki nie da się jednak wyrazić własności kolumn traktowanych jako całości, np. mówiących że kolumna jest uporządkowana, albo że nie zawiera powtórzeń. Dla wyrażenia takich własności trzeba zbudować specyficzne konstruktory dedykowane kolumnom. Jako przykład zdefiniuję jeden z nich zakładając, że pozostałe jarzma z tej grupy stanowią parametr modelu.

bez-powtórzeń-tb : Identyfikator $\mapsto$ Transfer

bez-powtórzeń-tb.ide.kom =

kom : Błąd $\rightarrow$ kom

rodzaj.kom $\neq$ 'Tq' $\rightarrow$ 'oczekiwana-tabela'

niech

kol = Km[weź-ko-tb].(ide, kom) (patrz rozdział 12.2.7)

kol : Błąd $\rightarrow$ kol

prawda $\rightarrow$ brak-powtórzeń.kol

Tworzymy krotkę kompozytów kol reprezentujący kolumnę tabeli o atrybucie ide, a następnie sprawdzamy, czy ta krotka spełnia uniwersalny predykat **brak-powtórzeń** (rozdział 2.1.4). Przypominam, że tworzona kolumna nie obejmuje elementów odpowiadających wierszowi wartości domyślnych.

Ponieważ wśród konstruktorów jarzm mamy (rozdział 5.2.4) konstruktory boolowskie, możemy ich użyć do tworzenia jarzm wyrażających własności wielu kolumn jednej tabeli w połączeniu z własnościami wszystkich jej wierszy. Zauważmy, że w odróżnieniu od standardu SQL, własności kolumn i wierszy możemy łączyć w dowolne kombinacje boolowskie, a nie jedynie koniunkcje¹²⁷.

Na koniec należy podkreślić, że w naszym modelu relacja podrzędności nie pojawia się na poziomie jarzm tabelowych, gdyż podrzędność tabeli wobec innej tabeli nie jest własnością tabeli, ale bazy danych. To spowoduje (rozdział 12.9), że SQL-owa deklaracja zmiennej tabelowej będzie w naszym przypadku deklaracją kolokwialną odpowiadające po stronie konkretnej sekwencji deklaracji zmiennej tabelowej i instrukcji bazodanowej.

12.5 Typy

Algebra typów **Lingua-SQL** zawiera cztery nośniki:

- Identyfikator
- Transfer
- KompozytB
- TypB

i powstaje z algebry typów dla **Lingua-A** przez jej rozszerzenie (patrz rozdział 2.11) o nośnik kompozytów i o trzy grupy konstruktorów:

1. wszystkie konstruktory transferów opisane w rozdziale 12.4,
2. wybrane konstruktory kompozytów wierszowych spośród opisanych w rozdziale 12.2,
3. konstruktory typów opisane poniżej.

¹²⁷ Prawdę mówiąc nie jestem pewien, czy takie uogólnienie znajdzie zastosowania praktyczne.

Obecność nośnika kompozytów w tej algebrze wynika z faktu, że do utworzenia typu tabelowego trzeba utworzyć kompozyt wierszowy wartości domyślnych dla kolumn. Jako konstruktory typów przyjmę (notacja analogicznie jak w rozdziale 5.2.5):

$Ky[\text{twórz-wi}] : \text{TypB} \times \text{Identyfikator} \mapsto \text{TypB}$

$Ky[\text{dołóż-do-wi}] : \text{TypB} \times \text{Identyfikator} \times \text{TypB} \mapsto \text{TypB}$

$Ky[\text{twórz-tb-pustą}] : \text{KompozytB} \times \text{Transfer} \mapsto \text{TypB}$

Typy wierszowe tworzymy analogicznie jak typy rekordowe (rozdział 5.2.5), z tym że tym razem dodawany typ musi być prosty.

Tworzenie typu wierszowego z jednym atrybutem

$Ky[\text{twórz-wi}] : \text{TypB} \times \text{Identyfikator} \mapsto \text{TypB}$

$Ky[\text{twórz-wi}].(\text{typ}, \text{ide}) =$

$\text{typ} : \text{Błąd} \rightarrow \text{typ}$

niech

$(\text{kor}, \text{tra}) = \text{typ}$

$\text{nowy-kor} = Kr[\text{twórz-wi}].(\text{ide}, \text{kor})$

$\text{nowy-tra} = Kt[\text{weź-z-wi}].\text{ide} \bullet \text{tra}$

nie $\text{kor} : \text{KorProsty} \rightarrow \text{'oczekiwany-typ-prosty'}$

prawda $\rightarrow (\text{nowy-kor}, \text{nowy-tra})$

Do klanu tak utworzonego typu należą wiersze, w których jedyna dana ma korpus kor i spełnia jarzmo tra (por. komentarz do definicji konstruktora $Ky[\text{twórz-re}]$ w rozdziale 5.2.5)

Dokładanie atrybutu do typu wierszowego

$Ky[\text{dołóż-do-wi}] : \text{TypB} \times \text{Identyfikator} \times \text{TypB} \mapsto \text{TypB}$

$Ky[\text{dołóż-do-wi}].(\text{typ-w}, \text{ide}, \text{typ-p}) =$ (w – wierszowy, p – prosty)

$\text{typ-i} : \text{Błąd} \rightarrow \text{typ-i}$ dla $i = w, p$

niech

$(\text{kor-w}, \text{tra-w}) = \text{typ-w}$

$(\text{kor-p}, \text{tra-p}) = \text{typ-p}$

$\text{rodzaj.kor-w} \neq \text{'Tq'} \rightarrow \text{'oczekiwany-typ-wierszowy'}$

nie $\text{kor-p} : \text{KorProsty} \rightarrow \text{'oczekiwany-typ-prosty'}$

$\text{kor-w.ide} = ! \rightarrow \text{'atrybut zajęty'}$

prawda $\rightarrow$

$(\text{dołóż-do-wi}.\text{kor-r}, \text{ide}, \text{kor-d}), \text{oraz-tr-K}.\text{tra-w}, Kt[\text{weź-z-wi}].\text{ide} \bullet \text{tra-p})$

Wiersze należące do klanu tego typu powstają przez rozbudowanie wierszy będących typu typ-w o nowy atrybut, którego dana ma korpus prosty kor-p i spełnia jarzmo tra-p.

Tworzenie typu tabelowego

$Ky[\text{twórz-tb-pustą}] : \text{KompozytB} \times \text{Transfer} \mapsto \text{TypB}$

$Ky[\text{twórz-tb-pustą}].(\text{kom}, \text{tra}) =$

$\text{kom} : \text{Błąd} \quad \rightarrow \text{kom}$

$\text{rodzaj.kom} \neq \text{'Wq'} \quad \rightarrow \text{'oczekiwany-wiersz'}$

prawda $\rightarrow (Kr[\text{twórz-tb-pustą}].\text{kom}, \text{tra})$

Definicja konstruktora $Kr[\text{twórz-tb-pustą}]$ jest zawarta implicite w definicji konstruktora $Km[\text{twórz-tb-pustą}]$ w rozdziale 12.2.6. Dla wygody czytelnika przytoczę ją jednak explicite:

$Km[\text{twórz-tb-pustą}].\text{kom} =$

$\text{kom} : \text{Błąd} \quad \rightarrow \text{kom}$

$\text{rodzaj.kom} \neq \text{'Wq'} \quad \rightarrow \text{'oczekiwany-wiersz'}$

niech

$(\text{wie}, \text{kor}) = \text{kom}$

prawda $\rightarrow (\text{'Tq'}, \text{wie}, \text{kor})$

Zatem typ tabelowy tworzony przez konstruktor $Ky[\text{twórz-tb-pustą}]$ jest postaci

$((\text{'Tq'}, \text{wie}, \text{kor}), \text{tra})$

Zauważmy, że typów bazodanowych nie wprowadzamy, co jest związane z faktem, że wartości bazodanowe (rozdział 12.6) nie będą parami składającymi się z kompozytu i typu.

12.6 Wartości bazodanowe

W dotychczasowych wersjach **Lingua** wartościami nazywaliśmy pary składające się z danej i jej typu. W **Lingua-SQL** takie rozumienie pojęcia wartości rozszerzamy na wartości SQL-owe proste, wierszowe i tabelowe, ale nie na wartości bazodanowe. Te ostatnie będą parami składającymi się z (intuicyjnie rozumianego) rekordu wartości tabelowych oraz grafu podrzędności (rozdział 12.2.2). W sprawie baz przyjmę dwa założenia:

- aby tabele bazy były dostępne dla programu, muszą być przypisane identyfikatorom zmiennych w wartościowaniu,
- w wartościowaniu będą w każdym momencie zapisane wartości tabelowe tylko jednej bazy, zwanej *bazą aktywną*.

Dla opisanego tego mechanizmu wprowadzę kilka nowych pojęć.

Zgodnie z wcześniejszą zapowiedzą w sposób oczywisty rozszerzam dotychczasową dziedzinę wartości prostych oraz wprowadzam dziedziny wartości wierszowych i tabelowych:

$\text{WarWie} = \{(\text{kom}, \text{tra}) \mid \text{rodzaj.kom} = \text{'Wq'} \text{ oraz } \text{tra.kom} = (\text{pp}, (\text{'boolowska'}))\}$

$\text{WarTab} = \{(\text{kom}, \text{tra}) \mid \text{rodzaj.kom} = \text{'Tq'} \text{ oraz } \text{tra.kom} = (\text{pp}, (\text{'boolowska'}))\}$

Rekordem bazodanowym nazwę mapping odwzorowujący identyfikatory w wartości tabelowe:

$$\text{rbd} : \text{RekBazDan} = \text{Identyfikator} \Rightarrow \text{WarTab}$$

Rekordy bazodanowe nie są oczywiście rekordami w sensie zdefiniowanym w rozdziale 5.2.1, a jedynie w sensie teoriomnogościowym.

O rekordzie bazodanowym *rbd* powiem, że *spełnia relacje podrzędności* wskazane przez *graf podrzędności* *grp*, co zapiszę formułą:

$$\text{rbd} \text{ spełnia } \text{grp}$$

gdy dla każdej krawędzi grafu (*ide-p*, *ide*, *ide-n*), dla której tabele

$$(\text{kom-p}, \text{tra-p}) = \text{rbd.ide-p}$$

$$(\text{kom-n}, \text{tra-n}) = \text{rbd.ide-n}$$

są określone, zachodzi relacja

$$\text{kom-p} \text{ pod}[\text{ide}] \text{ kom-n}$$

Wartością bazodanową nazwę każdą parę, składającą się z rekordu bazodanowego oraz grafu podrzędności wskazującego relacje spełniane przez ten rekord:

$$\text{wbd} : \text{WarBda} = \{(\text{rbd}, \text{grp}) \mid \text{rbd} \text{ spełnia } \text{grp}\}$$

Można powiedzieć, że w wartościach bazodanowych rolę jarzma odgrywa predykat spełniania. Zauważmy jednak, że ponieważ rekord bazodanowy niesie wartości tabelowe, to zawarte w bazie tabele spełniają odpowiadające im jarzma.

12.7 Algebra denotacji

Jak już kilkakrotnie zapowiadałem, **Lingua-SQL** będzie oferować wszystkie mechanizmy programistyczne **Lingua-3** i nadto mechanizmy odpowiadające wybranym narzędziom SQL-owym. W niniejszym rozdziale ograniczę się więc do zdefiniowania tego, co w **Lingua-SQL** jest nowe. Jednakże i tych mechanizmów nie będę opisywał ze wszystkimi szczegółami skupiając się przede wszystkim na pokazaniu metody.

12.7.1 Stany i dziedziny denotacyjne

Podobnie jak we wcześniejszych wersjach **Lingua** również w **Lingua-SQL** stany wiążą wartości ze zmiennymi danologicznymi oraz typy ze stałymi typologicznymi. Ogólne definicje typu i wartości pozostają takie jak w rozdziałach 5.2.5 oraz 5.3.1 z wyjątkiem wartości bazodanowych (rozdział 12.6). Zatem wartościami w **Lingua-SQL**, a więc tymi danymi utypowanymi, które mogą być przypisywane identyfikatorom zmiennych, są wszystkie wartości z poziomu **Lingua-3** oraz dodatkowo wartości niosące:

1. SQL-owe dane proste,
2. wiersze,
3. tabele,
4. bazy danych.

Oczywiście wartości bazodanowe nie są wartościami sensu stricte, gdyż nie składają się z danej i typu. Typ bazy danej jest implicite zawarty w typach jej tabel oraz w grafie podrzędności.

W każdym stanie może być zapisanych wiele baz danych, tj. wiele baz danych może być przypisanych identyfikatorom, jednakże tylko jedna baza może być aktywna, tj. tylko dla jednej bazy jej tabele mogą być przypisane identyfikatorom w wartościowaniu.

Pewnej modyfikacji podlega samo pojęcie stanu, gdyż zakładam istnienie w modelu czterech identyfikatorów systemowych:

- graf-rp — wiążący w środowisku typów graf podrzędności aktywnej bazy,
- kopie — wiążący w wartościowaniach skończony zbiór nazw tabel (identyfikatorów),
- monitor — wiążący w wartościowaniu jedną tabelę (aktualnie widoczną na monitorze)
- sprawdź — wiążący w wartościowaniach słowa 'tak' lub 'nie'.

których rola zostanie wyjaśniona nieco później. Na razie przyjmuję jedynie, że nie mogą być użyte jako identyfikatory zmiennych, typów i procedur. Przyjmę też, że **sprawdź** będzie nazywany *flagą bezpieczeństwa*. Jeżeli wartością flagi jest 'tak', to powiemy, że *flaga jest podniesiona*, a w przeciwnym przypadku, że *flaga jest opuszczona*.

Grafy podrzędności umieszczam w środowisku typologicznym, mimo że one nie są typami, bo nie składają się z korpusu i jarzma. Być może powinienem w tym miejscu wprowadzić nowy komponent stanu, który przechowywałby grafy podrzędności, podobnie jak to uczyniłem z komunikatami błędów. Zrezygnowałam jednak z tego rozwiązania, by pozostać przy definicji stanu z rozdziału 5.3.1, którą chcę traktować jako niezmiennik w moim modelu.

Sygnatura algebry denotacji języka **Lingua-SQL** jest rozszerzeniem sygnatury **Lingua** (rozdział 7.8.1.1 o nowe konstruktory. Nośniki powiększają się o nowe SQL-owe wartości i typy.

12.7.2 Denotacje wyrażeń danologicznych

W **Lingua-SQL** dopuszczam wszystkie denotacje wyrażeń danologicznych z **Lingua-3**, natomiast wyrażenia spoza **Lingua-3** są ograniczone do wyrażeń generujących kompozyty proste, wierszowe i tabelowe. Wyrażeń bazodanowych nie będzie, bo nie zostały wprowadzone bazodanowe kompozyty. Bazy danych pojawiają się więc dopiero na poziomie instrukcji (rozdział 12.7.6.11).

Zgodnie z przyjętymi założeniami, SQL-owe konstruktory denotacji wyrażeń danologicznych będą pochodnymi SQL-owych konstruktorów kompozytów. Przypominam więc (rozdział 5.3.2), że *Kdd* oznacza operator, który konstruktory denotacji wyrażeń tworzy z konstruktorów kompozytów zgodnie ze schematem (5.3.2-1).

W przypadku tabel założę dodatkowo, że w wyrażeniach tabelowych przyjmujących tabele jako argumenty, te ostatnie mogą być reprezentowane jedynie przez identyfikatory, a nie przez wyrażenia złożone. Na przykład konstruktorowi kompozytów tabelowych:

$Km[do\acute{l}\acute{o}ż-wi-do-tb] : KompozytB \times KompozytB \mapsto KompozytB$

będzie odpowiadał konstruktor denotacji wyrażeń:

$Kdd[Km[do\acute{l}\acute{o}ż-wi-do-tb]] : DenWyrDan \times Identyfikator \mapsto DenWyrDan$

$Kdd[Km[do\acute{l}\acute{o}ż-wi-do-tb]].(dwd, ide).sta =$

$Km[do\acute{l}\acute{o}ż-wi-do-tb].(dwd.sta, zmienna-dan.ide.sta)$

To oczywiście decyzje inżynierska, a nie matematyczna konieczność. Przyjmuję ją, gdyż upraszcza analizę składniową wyrażeń i jest też chyba zgodna ze standardem SQL.

Ponieważ definicje wszystkich konstruktorów wyrażeń danologicznych podpadają pod wspólny schemat, nie będę ich tu przytaczał. Pokażę jedynie sygnatury tych konstruktorów, gdyż będą one potrzebne do wygenerowania składni **Lingua-SQL**.

Konstruktory denotacji wyrażeń generujących kompozyty proste

Te konstruktory traktuję jako parametry modelu, zakładam jednak obecność specyficznych konstruktorów generujących wyrażenia tworzące kompozyty puste. Dla każdego korpusu prostego $\text{kor} : \text{KorProsty}$ zakładam istnienie zeroargumentowego konstruktora denotacji:

$\text{Kdd}[\text{Km}[\text{puste.kor}]] : \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{puste.kor}]].\text{sta} =$

$\text{jest-błąd.sta} \rightarrow \text{błąd.sta}$

prawda $\rightarrow (\emptyset, \text{kor})$

Konstruktory denotacji wyrażeń wierszowych

$\text{Kdd}[\text{Km}[\text{twórz-wi}]] : \text{Identyfikator} \times \text{DenWyrDan} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{dołącz-do-wi}]] : \text{Identyfikator} \times \text{DenWyrDan} \times \text{DenWyrDan} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{usuń-z-wi}]] : \text{Identyfikator} \times \text{DenWyrDan} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{weź-z-wi}]] : \text{Identyfikator} \times \text{DenWyrDan} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{zmień-w-wi}]] : \text{Identyfikator} \times \text{DenWyrDan} \times \text{Transfer} \mapsto \text{DenWyrDan}$

Wierszowe konstruktory denotacji wyrażeń tabelowych

$\text{Kdd}[\text{Km}[\text{twórz-tb-pusta}]] : \text{DenWyrDan} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{dołącz-wi-do-tb}]] : \text{DenWyrDan} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{usuń-wi-z-tb}]] : \text{Transfer} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{weź-wi-z-tb}]] : \text{Transfer} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{wyklucz-wi-z-tb}]] : \text{Identyfikator} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{filtruj-wi-w-tb}]] : \text{Transfer} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{połącz-tb}]] : \text{Identyfikator} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{przetnij-tb}]] : \text{Identyfikator} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

Kolumnowe konstruktory denotacji wyrażeń tabelowych

$\text{Kdd}[\text{Km}[\text{dołącz-ko-do-tb}]] : \text{Identyfikator} \times \text{DenWyrDan} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{usuń-ko-z-tb}]] : \text{Identyfikator} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{filtruj-ko-z-tb}]] : \text{ParAkt} \times \text{Identyfikator} \mapsto \text{DenWyrDan}$

$\text{Kdd}[\text{Km}[\text{zmień-ko-w-tb}]] : \text{Identyfikator} \times \text{Identyfikator} \times \text{Transfer} \times \text{Transfer}$
 $\mapsto \text{KompozytB}$

Konstruktor denotacji wyrażenia tworzącego tabelę pochodną

$\text{Kdd}[\text{twórz-tb-ref}] : \text{Identyfikator} \times \text{Identyfikator} \times \text{Identyfikator} \times \text{Transfer}$
 $\mapsto \text{DenWyrDan}$

Zauważmy, że w tym konstruktorze nie występuje $\text{Km}[\dots]$ gdyż twórz-tb-ref jest konstruktorem kompozytów (rozdział 12.2.8).

12.7.3 Denotacje wyrażen typologicznych i transferowych

Przypominam (rozdział 5.3.4), że algebra denotacji wyrażen danologicznych, typologicznych i transferowych w **Lingua-3** obejmuje cztery nośniki:

$\text{Ide} : \text{Identyfikator}$
 $\text{dwd} : \text{DenWyrDan} = \text{Stan} \rightarrow \text{KompozytB}$
 $\text{tra} : \text{DenWyrTra} = \text{Transfer},$
 $\text{dwt} : \text{DenWyrTyp} = \text{Stan} \mapsto \text{TypB}$

W **Lingua-SQL** ta sytuacja pozostaje bez zmian.

Zgodnie z zasadami opisanymi w rozdziale 5.3.5, transfery są denotacjami wyrażen transferowych. Jak już wyjaśniałem, to decyzja inżynierska, która powoduje, że transferów nie przypisujemy identyfikatorom w stanach, a więc nie zapamiętujemy.

Do konstruktorów dostępnych w **Lingua-3** dochodzą więc:

- konstruktory transferów SQL-owych (rozdział 12.4),
- konstruktory denotacji wyrażen typologicznych będące pochodnymi konstruktorom typów SQL-owych (rozdział 12.5).

Te założenia prowadzą do poniższej listy konstruktorów, które obejmuje algebra denotacji dla **Lingua-SQL**. Przypominam (rozdział 5.3.4), że Kdt jest konstruktorem, który z konstruktorów typologicznych i transferowych tworzy konstruktory denotacji

Konstruktory denotacji wyrażen transferowych

$\text{Kt}[\text{weź-z-wi}] : \text{Identyfikator} \mapsto \text{Transfer}$
 $\text{bez-powtórzeń-tb} : \mapsto \text{Transfer}$
 $\text{wszystkie-w-tb} : \text{Transfer} \mapsto \text{Transfer}$

Konstruktory denotacji wyrażen typologicznych

$\text{Kdt}[\text{Ky}[\text{twórz-wi}]] : \text{DenWyrTyp} \times \text{Identyfikator} \mapsto \text{DenWyrTyp}$
 $\text{Kdt}[\text{Ky}[\text{dołóż-do-wi}]] : \text{DenWyrTyp} \times \text{Identyfikator} \times \text{DenWyrTyp} \mapsto \text{DenWyrTyp}$
 $\text{Kdt}[\text{Ky}[\text{twórz-tb-pustą}]] : \text{DenWyrDan} \times \text{Transfer} \mapsto \text{DenWyrTyp}$

Konstruktory pierwszej grupy zostały już definiowane w rozdziale 12.4, dwa kolejne konstruktory definiujemy zgodnie z zasadami opisanymi w rozdziale 5.3.4. Konstruktor tworzenia tabeli pustej wymaga odrębnej definicji, gdyż korpus tabelowy (a więc i typ) obejmuje nie tylko korpusy przypisane atrybutom, ale też ich wartości domyślne. Stąd wśród argumentów tego konstruktora musi się znaleźć denotacja wyrażenia danologicznego:

```
Kdt[Ky[twórz-tb-pustą]].(dwd, tra).sta =
  jest-błąd.sta    → błąd.sta
  dwd.sta = ?      → ?
  let
    kom = dwd.sta
  prawda          → Ky[twórz-tb-pustą].(kom, tra)
```

12.7.4 Denotacje definicji stałych typologicznych

Nowe definicje stałych typologicznych w **Lingua-SQL** w stosunku do **Lingua-3** to definicje odwołujące się do nowych konstruktorów typów (rozdział 12.5), które podpadają pod ogólny schemat opisany w rozdziale 6.1.3, nie ma więc powodu, by je tu powtarzać.

Konstruktory związane z budowaniem i modyfikowaniem grafów podrzędności nie należą do tej grupy, gdyż w wyniku ich „działania” nie powstaje żadna stała typologiczna, a jedynie zmienia się graf podrzędności przypisany identyfikatorowi systemowemu graf-rp. Te konstruktory pojawią się więc dopiero na poziomie instrukcji bazodanowych w rozdziale 12.7.6.11.

Konstruktor definicji związanej z grafami podrzędności jest jeden:

```
dodaj-pod-typ : Identyfikator x Identyfikator x Identyfikator → DenDefTyp
dodaj-pod-typ.(ide-p, ide, ide-n).sta =
  jest-błąd.sta          → sta
  ide-p = ide-n           → sta ◀ ‘zwrotność-niedopuszczalna’
  niech
    ((śrt, śrp), skł) = sta
    grp = śrt.graf-rp
  (ide-p, ide, ide-n) : grp → ‘redundantna-podrzędność’
  niech
    nowy-grp = grp | {(ide-p, ide, ide-n)}
  prawda                 → ((śrt[graf-rp/nowy-grp], śrp), skł)
```

Ten konstruktor powiększa graf podrzędności o nową krawędź i uaktualnia środowisko typologiczne.

Na poziomie definicji typów nie wprowadzam konstruktora usuwania krawędzi z grafu, bo w naszym modelu (tj. we wszystkich językach z grupy **Lingua**) typy, w odróżnieniu od wartości, nie są modyfikowalne. Mówiąc bardziej formalnie, każda stała typologiczna jest definiowana w każdym programie tylko raz, a przypisany jej typ nie jest później zmieniany. Może natomiast zmieniać się typ danej związanej ze zmienną danologiczną. W wersjach **Lingua** od

1 do 3 dotyczy to zmiennych rekordowych, a w **Lingua-SQL** będzie również dotyczyło zmiennych wierszowych i tabelowych. Usuwanie krawędzi z grafu podrzędności pojawi się więc dopiero na poziomie instrukcji bazodanowych w rozdziale 12.7.6.11.

12.7.5 Denotacje deklaracji zmiennych danologicznych

W **Lingua-SQL** zmienne przyjmują wszystkie wartości dostępne w **Lingua-3**, a dodatkowo cztery rodzaje specyficznych wartości SQL-owych:

1. proste wartości SQL-owe,
2. wartości wierszowe,
3. wartości tabelowe,
4. wartości bazodanowe.

Deklaracje zmiennych pierwszych dwóch grup podpadają pod ogólny schemat konstruktora deklaracji zmiennych w **Lingua-3** (rozdział 6.1.2). Deklaracji danologicznych zmiennych bazodanowych nie będzie, a w ich miejsce pojawi się specyficzna instrukcja archiwizowania bazy przez przypisanie jej do wskazanego identyfikatora (rozdział 12.7.6.11). Konstruktor deklaracji zmiennej tabelowej jest zdefiniowany w sposób nieco odbiegający od standardu dla zmiennych danologicznych **Lingua-3**:

deklaruj-zmi-tab : Identyfikator $\times$ DenWyrTyp $\mapsto$ DenDekZmi

deklaruj-zmi-tab.(ide, dwt).sta =

jest-błąd.sta $\rightarrow$ sta

zadeklarowany.ide.sta $\rightarrow$ sta $\blacktriangleleft$ 'identyfikator-zajęty'

niech

typ = dwt.sta

typ : Błąd $\rightarrow$ sta $\blacktriangleleft$ typ

niech

(kor, jar) = typ

rodzaj.kor $\neq$ 'Tq' $\rightarrow$ sta $\blacktriangleleft$ 'oczekiwany-typ-tabelowy'

niech

war = ((), typ)

(śro, (wrt, 'OK')) = sta

prawda $\rightarrow$ (śro, (wrt[ide/war], 'OK'))

Różnica pomiędzy tą definicją, a naszym standardem z rozdziału 6.1.2 jest przede wszystkim taka, że tym razem zmiennej przypisujemy tabelę pustą ((), typ), a nie pseudowartość (Ω , typ). Następuje też sprawdzenie, czy typ jest typem tabelowym.

12.7.6 Instrukcje

12.7.6.1 Kategorie instrukcji SQL-owych

Nośnik denotacji instrukcji w algebrze denotacji dla **Lingua-SQL** otrzymuje nowe konstruktory związane ze specyficznymi instrukcjami SQL-owymi trzech kategorii:

1. przypisania wierszowe,
2. przypisania tabelowe,
3. instrukcje bazodanowe.

Wszystkie zdefiniowane wcześniej konstruktory występujące w **Lingua-3** pozostają nadal dostępne i stosują się do rozszerzonego nośnika denotacji instrukcji. Dotyczy to w szczególności konstruktora wymiany transferu oraz konstruktorów instrukcji strukturalnych, a więc składania sekwencyjnego, rozgałęzienia oraz pętli. Nie zmieniają się też konstruktory deklarowania i wołania procedur poza tym, że powiększają się dziedziny, na których są zdefiniowane.

Szczególną rolę w SQL odgrywa liczna grupa przypisań tabelowych, wśród których wyróżnię dwie kategorie:

1. *instrukcje modyfikacji tabeli*, gdzie po obu stronach przypisania stoi nazwa tej samej tabeli,
2. *instrukcje tworzenia tabeli*, gdzie po lewej stronie przypisania może stać nazwa innej (tej tworzonej) tabeli, niż po prawej.

Matematycznie pierwszy z tych rodzajów można uznać za szczególny przypadek drugiego, ale denotacyjne będą one odpowiadały różnym konstruktorom algebry denotacji, a więc i różnym konstruktorom algebry składni. Przyczyna takiego rozwiązania zostanie wyjaśniona w kolejnych rozdziałach.

Niezależnie od przedstawionego wyżej podziału, przypisania tabelowe dzielę na kolejne dwa rodzaje ze względu na sposób wykorzystywania więzów podrzędności (por. rozdział 11.5):

1. *instrukcje zachowawcze*, których wykonanie kończy się komunikatem błędu, gdyby miało nastąpić naruszenie więzów podrzędności; te instrukcje odpowiadają opcji **RESTRICT** opisanej w rozdziale 11.5,
2. *instrukcje korygujące*, które w opisanej powyżej sytuacji wprowadzają do bazy zmiany gwarantujące zachowanie więzów podrzędności; te instrukcje odpowiadają opcji **CASCADE** opisanej w rozdziale 11.5.

O ile dobrze się doczytałem w podręcznikach cytowanych na początku rozdziału 11.1, pierwsza z tych opcji jest (najczęściej?) domyślna, a druga musi zostać zadeklarowana *explicite* i jest dostępna jedynie dla niektórych instrukcji, np. przy usuwaniu wiersza lub kolumny z tabeli.

12.7.6.2 Instrukcje wierszowe

Instrukcje wierszowe tworzą i modyfikują wartości wierszowe przypisywane identyfikatorom w stanach. Budujemy je posługując się konstruktorem przypisania zdefiniowanym w rozdziale 6.1.4 oraz konstruktorami denotacji wyrażeń danologicznych zwracających wartości wierszowe, które zostały opisane w rozdziale 12.7.2.

W tym miejscu jest konieczna jedna uwaga techniczna. Otóż aby móc zastosować zdefiniowany wcześniej konstruktor przypisania, trzeba na korpusy wierszowe i tabelowe rozszerzyć zdefiniowaną tamże relację koherentności. To jednak proste zadanie, gdyż korpusy zarówno

wierszy jak i tabel mają strukturę rekordu. Założmy więc, że od tej pory relacja koherentny obejmuje wiersze i tabele.

12.7.6.3 Dwa uniwersalne konstruktory przypisań tabelowych

W **Lingua-SQL** mamy dwa konstruktory przypisań odpowiadające odpowiednio przypisaniu tabeli do zmiennej tabelowej oraz przypisaniu tabeli do zmiennej systemowej monitor.

W pierwszym przypadku można by zastosować uniwersalny konstruktor opisany w rozdziale 6.1.4 gdyby nie fakt, że przypisanie może prowadzić do takiej modyfikacji istniejącej tabeli, które naruszy bazodanową relację podrzędności. Trzeba więc wprowadzić specyficzny konstruktor przypisania dla tabel obejmujący ogólny przypadek ewentualnego naruszania relacji podrzędności. Jak zobaczymy dalej, stanie się on wygodnym narzędziem do definiowania wielu instrukcji przypisań tabelowych. Ponieważ jednak takiego konstruktora nie ma w standardzie SQL, jego udostępnienie na poziomie składni **Lingua-SQL** pozostawiam na razie sprawą otwartą.

Drugi uniwersalny konstruktor przypisań tabelowych posłuży do definiowania denotacji zapytań.

Dla zdefiniowania pierwszego z tych konstruktorów wprowadzę wpierw dwie funkcje pomocnicze, które nazwę *funkcjami kontroli naruszeń*.

Pierwsza z tych funkcji służy do sprawdzenia, czy zadany identyfikator wskazuje w aktualnym stanie na tabelę, która narusza którąkolwiek z aktualnie zadeklarowanych relacji podrzędności.

$\text{naruszona-rp} : \text{Identyfikator} \times \text{Stan} \mapsto \{\text{tt}, \text{ff}\} \mid \text{Błąd}$

$\text{naruszona-rp}(\text{ide}, \text{sta}) =$

$\text{jest-błąd.sta} \quad \rightarrow \text{błąd.sta}$

niech

$((\text{śrp}, \text{śrt}), (\text{wrt}, \text{'OK'})) = \text{sta}$

$\text{grp} = \text{śrt.graf-rp}$

$\text{wrt.ide} = ? \quad \rightarrow \text{'nie-ma-takiej-tabeli'}$

$\text{rodzaj}(\text{wrt.ide}) \neq \text{'Tq'} \quad \rightarrow \text{'oczekiwana-tabela'}$

$(\exists (\text{ide-p}, \text{ide-id}, \text{ide-n}) : \text{grp})$

$[\text{wrt.ide-i} = ! \text{ oraz } \text{rodzaj}(\text{wrt.ide-i}) = \text{'Tq'}] \text{ dla } i = p, n \text{ oraz}$

$((\text{ide} = \text{ide-p} \text{ oraz nie } \text{wrt.ide pod}[\text{ide-id}] \text{ wrt.ide-p}) \text{ lub}$

$(\text{ide} = \text{ide-n} \text{ oraz nie } \text{wrt.ide-p pod}[\text{ide-id}] \text{ wrt.ide}))$

$\rightarrow \text{tt}$

prawda

$\rightarrow \text{ff}$

Zatem ta funkcja przyjmuje wartość tt, gdy tabela wrt.ide nie spełnia warunku podrzędności wyznaczonego przez krawędź (ide, ide-id, ide-n) lub przez krawędź (ide-p, ide-id, ide).

Druga funkcja jest podobna do pierwszej i służy do sprawdzenia, czy zadany kompozyt spełnia zadane jarzmo tabelowe. Zauważmy, że sprawdzenie może być wyłączone przez ustawienie

flagi sprawdź na ‘nie’. W ten właśnie sposób jest realizowany mechanizm chwilowego wyłączania sprawdzeń więzów spójności opisywanych jarzmami. Zwracam uwagę, że dla więzów opisywanych relacjami podrzędności takiej opcji nie wprowadziłem¹²⁸.

$\text{naruszone-ja} : \text{Kompozyt} \times \text{Transfer} \times \text{Stan} \mapsto \text{Transfer}$

$\text{naruszone-ja}(\text{kom}, \text{tra}, \text{sta}) =$

$\text{jest-błąd.sta} \quad \rightarrow \text{błąd.sta}$

niech

$(\text{śro}, (\text{wrt}, \text{'OK'})) = \text{sta}$

$\text{wrt.sprawdź} = \text{'nie'} \quad \rightarrow \text{ff} \quad (\text{sprawdź jest ident. system.}; \text{rozdział 12.7.1})$

$\text{tra.kom} = (\text{pp}, (\text{'boolowska'})) \quad \rightarrow \text{ff}$

prawda $\quad \rightarrow \text{tt}$

Jak więc widzimy, sprawdzenie, czy jarzmo tra zostało naruszone, dokonuje się jedynie wtedy, gdy flaga jest podniesiona (ustawiona na ‘tak’). Zwróćmy też uwagę, że ta funkcja zwraca tt (sygnalizuje naruszenie jarzma) również wtedy gdy wynik sprawdzenia tra.kom jest błędem. Teraz możemy zdefiniować konstruktor przypisań dla tabel.

$\text{przypisz-tb} : \text{Identyfikator} \times \text{DenWyrDan} \mapsto \text{DenIns}$

$\text{przypisz-tb}(\text{ide}, \text{dwd}).\text{sta} =$

$\text{jest-błąd.sta} \quad \rightarrow \text{sta}$

$\text{wrt.ide} = ? \quad \rightarrow \text{sta} \blacktriangleleft \text{'niezadeklarowany-identyfikator'}$

$\text{dwd.sta} = ? \quad \rightarrow ?$

$\text{dwd.sta} : \text{Błąd} \quad \rightarrow \text{sta} \blacktriangleleft \text{dwd.sta}$

niech

$((\text{śrt}, \text{śrp}), (\text{wrt}, \text{'OK'})) = \text{sta}$

$\text{kom-n} = \text{dwd.sta}$

$(\text{kom-d}, \text{jar}) = \text{wrt.ide}$

$(\text{dan-n}, \text{kor-n}) = \text{kom-n} \quad (\text{n} - \text{nowa})$

$(\text{dan-d}, \text{kor-d}) = \text{kom-d} \quad (\text{d} - \text{dawna})$

$\text{rodzaj.kor-n} \neq \text{'Tq'} \quad \rightarrow \text{sta} \blacktriangleleft \text{'oczekiwana-tabela'}$

$\text{rodzaj.kor-d} \neq \text{'Tq'} \quad \rightarrow \text{sta} \blacktriangleleft \text{'oczekiwana-tabela'}$

nie kor-n koherentny $\text{kor-d} \quad \rightarrow \text{sta} \blacktriangleleft \text{'brak-koherencji'}$

¹²⁸ Przyjmuję, że gdyby relacja podrzędności miała być chwilowo naruszona, np. pomiędzy usunięciem jednej kolumny a wstawieniem drugiej, to programista powinien wpisać do programu dwie odpowiednie instrukcje modyfikacji grafu podrzędności. Taka zasada zapewnia lepsze rozumienie działania programu, a także ułatwia zastosowanie odpowiednich reguł budowania instrukcji specyfikowanych.

niech

$$\text{sta-n} = (\text{śro}, (\text{wrt}[\text{ide}/(\text{kom-n}, \text{jar})], \text{'OK'}))$$

$$\text{naruszone-ja.}(\text{kom-n}, \text{jar}, \text{sta-n}) \rightarrow \text{sta} \leftarrow \text{'naruszone-jarzmo-tabelowe'}$$

$$\text{naruszona-rp.}(\text{kom-n}, \text{sta-n}) \rightarrow \text{sta} \leftarrow \text{'naruszona-relacja-podrzędności'}$$

$$\text{prawda} \rightarrow \text{stan-n}$$

W wykonaniu takiego przypisania identyfikatorowi *ide* zostaje w miejsce dawnego kompozytu *kom-d* przypisany nowy *kom-n* pod warunkiem, że:

1. oba są tabelowe i wzajemnie koherentne,
2. nowy kompozyt spełnia w nowym stanie zastane jarzmo *jar*, o ile tylko flaga sprawdzania nie jest ustawiona na 'nie',
3. nowy kompozyt nie zaburza w nowym stanie aktualnej relacji podrzędności.

Zauważmy, że zaburzenie relacji podrzędności może nastąpić tylko wtedy, gdy przypisanie modyfikuje istniejącą tabelę.

Drugi specyficzny konstruktor przypisań odpowiada sytuacji, gdy tabelę opisaną wyrażeniem tabelowym przypisujemy do systemowego konstruktora *monitor*, co fizycznie powoduje, że zostaje ona wyświetlona na monitorze. W tym przypadku na tworzoną tabelę nie nakładamy żadnego jarzma, co wyrażamy przez fakt, że jarzmem tak tworzonej wartości tabelowej jest *PP*.

$$\text{przypisz-mo} : \text{DenWyrDan} \mapsto \text{DenIns}$$

$$\text{przypisz-mo.dwd.sta} =$$

$$\text{jest-błąd.sta} \rightarrow \text{sta}$$
niech

$$(\text{śro}, (\text{wrt}, \text{'OK'})) = \text{sta}$$

$$\text{kom} = \text{dwd.sta}$$

$$\text{kom} : \text{Błąd} \rightarrow \text{sta} \leftarrow \text{kom}$$

$$\text{rodzaj.kom} \neq \text{'Tq'} \rightarrow \text{'oczekiwana-tabela'}$$

$$\text{prawda} \rightarrow (\text{śro}, (\text{wrt}[\text{monitor}/(\text{kom}, \text{PP})], \text{'OK'}))$$

Jak widzimy, w tym przypadku nie oczekujemy, aby identyfikator *monitor* był zadeklarowany. Jako identyfikator systemowy jest on zawsze dostępny i może przyjąć dowolną wartość tabelową. Jeżeli w chwili wykonywania opisanego przypisania jest z nim już związana jakaś wartość tabelowa, to zostaje ona nadpisana nową wartością.

12.7.6.4 Transakcje

Transakcje, podobnie jak instrukcje, są transformacjami stanów, jednakże w odróżnieniu od tych pierwszych są to funkcje całkowite, bo nie będzie wśród nich pętli i wywołań procedur. Ponadto, nie tworzą nowych tabel, a jedynie modyfikują istniejące. Zatem ich dziedzina jest następująca:

$$\text{dtr} : \text{DenTrn} = \text{Stan} \mapsto \text{Stan}$$

Denotacje transakcji grupuję jednak w odrębny nośnik algebry denotacji, by nie wszystkie instrukcje mogły być zastosowane w kontekstach przewidzianych dla transakcji. To oczywiście spowoduje, że transakcje będą również odrębną kategorią składniową.

Największą grupę transakcji stanowią *modyfikacje tabel*. Są to przypisania, które w tradycyjnej składni mogłyby mieć postać:

```
ide := wyrażenie-tabelowe(ide)
```

gdzie po lewej i prawej stronie stoi nazwa tej samej tabeli. Denotacje tych przypisań powstają w wyniku kombinacji przypisania tabelowego (rozdział 12.7.6.3) oraz którejś z denotacji wyrażenia tabelowego lub transferowego. Oto konstruktory denotacji takich transakcji, które odwołują się do denotacji wyrażeń danologicznych opisanych w rozdziale 12.7.2. Pierwszy z nich odpowiada dołożeniu wiersza do tabeli:

$\text{dołoż-wi} : \text{Identyfikator} \times \text{DenWyrDan} \mapsto \text{DenTrn}$

$\text{dołoż-wi}(\text{ide}, \text{dwd-w}) =$

$\text{przypisz-tb}(\text{ide}, \text{Kdd}[\text{Km}[\text{dołoż-wi-do-tb}]].(\text{zmienna-dan.ide}, \text{dwd-w}))$

W wyniku wykonania tego konstruktora powstaje denotacja transakcji, która do tabeli niesionej przez identyfikator *ide* dokłada wiersz wskazany przez denotację *dwd-w*. Rozczytajmy definicję tego konstruktora dokładniej:

Konstruktor przypisania tabelowego *przypisz-tb* jako pierwszy argument otrzymuje identyfikator wskazujący na modyfikowaną tabelę, z jako drugi — denotację wyrażenia powstającą za pomocą konstruktora $\text{Kdd}[\text{Km}[\text{dołoż-wi-do-tb}]]$, któremu jako argumenty zadano dwie denotacje wyrażeń:

- denotację wyrażenia będącego zmienną, a więc *zmienna-dan.ide*, które wskazuje na modyfikowaną tabelę,
- denotację wyrażenia wierszowego *dwd-w*, wskazującą na wiersz, który ma być dodany do tabeli.

W analogiczny sposób można zdefiniować kolejne konstruktory związane z modyfikacjami tabel:

$\text{usuń-wi} : \text{Identyfikator} \times \text{Transfer} \mapsto \text{DenTrn}$

$\text{usuń-wi}(\text{ide}, \text{tra}) =$

$\text{przypisz-tb}(\text{ide}, \text{Kdd}[\text{Km}[\text{usuń-wi-z-tb}]].(\text{tra}, \text{zmienna-dan.ide}))$

$\text{wyklucz-wi} : \text{Identyfikator} \times \text{identyfikator} \mapsto \text{DenTrn}$

$\text{wyklucz-wi}(\text{ide-1}, \text{ide-2}) =$

$\text{przypisz-tb}(\text{ide-1}, \text{Kdd}[\text{Km}[\text{wyklucz-wi-z-tb}]].(\text{zmienna-dan.ide-1},$
 $\text{zmienna-dan.ide-2}))$

$\text{dołoż-ko} : \text{Identyfikator} \times \text{DenWyrDan} \times \text{Identyfikator} \mapsto \text{DenTrn}$

$\text{dołoż-ko}(\text{ide-k}, \text{dwd}, \text{ide-t}) =$ (k – kolumna, t – tabela)

$\text{przypisz-tb}(\text{ide-t}, \text{Kdd}[\text{Km}[\text{dołoż-ko-do-tb}]].(\text{ide-k}, \text{dwd}, \text{zmienna-dan.ide-t}))$

$\text{usuń-ko} : \text{Identyfikator} \times \text{Identyfikator} \mapsto \text{DenTrn}$

$\text{usuń-ko}(\text{ide-k}, \text{ide-t}) =$

$\text{przypisz-tb}(\text{ide-t}, \text{Kdd}[\text{Km}[\text{usuń-ko-z-tb}](\text{ide-k}, \text{zmienna-dan.ide-t})])$

$\text{filtruj-ko} : \text{ParAkt} \times \text{Identyfikator} \mapsto \text{DenTrn}$

$\text{filtruj-ko}(\text{pak}, \text{ide}) =$

$\text{przypisz-tb}(\text{ide}, \text{Kdd}[\text{Km}[\text{filtruj-ko-z-tb}](\text{pak}, \text{zmienna-dan.ide})])$

$\text{zmień-ko} : \text{Identyfikator} \times \text{Identyfikator} \times \text{Transfer} \mapsto \text{DenTrn}$

$\text{zmień-ko}(\text{ide-k}, \text{ide-t}, \text{tra}) =$

$\text{przypisz-tb}(\text{ide-t}, \text{Kdd}[\text{Km}[\text{zmień-ko-w-tb}](\text{ide-k}, \text{zmienna-dan.ide-t}, \text{tra})])$

Drugą grupę transakcji stanowią *polecenia ochrony* (pojęcie własne), które służą do zabezpieczenia bazy przed zniszczenia.

Twórz kopię bezpieczeństwa

$\text{twórz-kopię-bezp} : \text{Identyfikator} \mapsto \text{Stan} \mapsto \text{Stan}$

$\text{twórz-kopię-bezp.sta} =$

$\text{jest-błąd.sta} \rightarrow \text{sta}$

niech

$(\text{śro}, (\text{wrt}, \text{'OK'})) = \text{sta}$

$\text{baza} = \text{wrt ogr} \{ \text{ide} \mid \text{wrt.ide} : \text{WarTab} \}$

$\text{grp} = \text{śrt.graf-rp}$

$\text{wykaz-kopii} = \text{wrt.kopie} \mid \{ \text{ide} \}$

$\text{wrt.ide} = ! \rightarrow \text{'identyfikator-zajęty'}$

prawda $\rightarrow (\text{śro}, (\text{wrt}[\text{ide}/(\text{baza}, \text{grp})], \text{kopie}/\text{wykaz-kopii}], \text{'OK'})$

Funkcja tworzy bazę ze wszystkich wartości tabelowych aktualnie występujących w wartościowaniu oraz aktualnego grafu podrzędności, a następnie przypisuje ją do identyfikatora *ide*. Do zbioru identyfikatorów kopii przypisanych do identyfikatora systemowego *kopie* dodaje identyfikator *ide*.

Przypominam (rozdział 2.1.3), że *ogr* oznacza operator ograniczenie funkcji do wskazanego podzbioru jej dziedziny.

Usuń kopię bezpieczeństwa

$\text{usuń-kopię-bezp} : \text{Identyfikator} \mapsto \text{DenTrn}$

```
usuń-kopię-bezp.ide.sta =
  jest-błąd.sta      → sta
niech
  (śro, (wrt, 'OK')) = sta
  wykaz-kopii      = wrt.kopie – {ide}
wrt.ide = ?          → 'identyfikator-nieznany'
nie wrt.ide : WarBda → 'oczekiwana-baza-danych'
prawda              → (śro, (wrt[ide/?, kopie/wykaz-kopii], 'OK'))
```

Kopia bazy jest usuwana z wartościowania, a jej nazwa z wykazu kopii.

Przywróć kopię bezpieczeństwa

```
przywróć-kopię-bezp : Identyfikator ↦ DenTrn
przywróć-kopię-bezp.ide.sta =
  jest-błąd.sta      → sta
niech
  (śro, (wrt, 'OK')) = sta
  wrt.ide = ?          → 'identyfikator-nieznany'
nie wrt.ide : WarBda → 'oczekiwana-baza-danych'
niech
  (rbd, grp)      = wrt.ide
  wykaz-kopii    = wrt.kopie – {ide}
prawda          → (śro, (wrt[ide/?, kopie/wykaz-kopii] ♦ rbd, 'OK'))
```

Niesiona przez identyfikator *ide* baza danych *rbd* jest mappingiem przypisującym identyfikatorom wartości tabelowe. Tym mappingiem przesłaniamy aktualne wartościowanie, z którego przedtem została usunięta baza niesiona przez *ide*. Nazwę usuwanej kopii usuwamy z wykazu kopii.

Przywróć kopię bezpieczeństwa warunkowo

```
przywróć-kopię-bezp-gdy : DenWyrDan x Identyfikator ↦ DenTrn
przywróć-kopię-bezp-gdy.(dwd, ide) = jeżeli-błąd.(dwd, przywróć-kopię-bezp.ide)
```

Gdy jako błąd pojawi się wskazany przez *dwd* komunikat błędu, to jest wykonywana procedura przywracania kopii bezpieczeństwa. W tym przypadku odwołuję się do zdefiniowanego w rozdziale 6.1.6 konstruktora obsługi błędów *jeżeli-błąd*. To odwołanie jest nieco nieformalne, bo drugim argumentem konstruktora *jeżeli-błąd* powinna być denotacja instrukcji, a w naszym

przypadku jest to denotacja transakcji. Jednakże teoriomnogościowo denotacje transakcji należą do dziedziny denotacji instrukcji, a więc nasza definicja ma sens.

Kolejne dwa konstruktory tworzą transakcje pozwalające na ustawianie wartości flagi odpowiadającej identyfikatorowi systemowemu **sprawdź**:

Opuść flagę sprawdzania

wyłącz-sprawdzanie : $\mapsto$ DenTrn

wyłącz-sprawdzanie().sta =

jest-błąd.sta $\rightarrow$ sta

niech

(śro, (wrt, 'OK')) = sta

wrt.sprawdź = 'nie' $\rightarrow$ 'sprawdzanie-jest-wyłączone'

prawda $\rightarrow$ (śro, (wrt[sprawdź/'nie'], 'OK'))

Ten konstruktor sygnalizuje błąd, gdy próbujemy ustawić na 'nie' flagę, które już jest tak ustawiona. To oczywiście nie konieczność matematyczna, ale decyzja inżynierska mająca chronić programistę przed ewentualnym popełnieniem błędu. Skoro bowiem umieszcza on opuszczenie flagi w kontekście, w którym jest ona opuszczona, to oznacza, że być może nie do końca ogarnia funkcjonalność programu. Drugi konstruktor z tej grupy podnosi flagę:

Podnieś flagę sprawdzania

włącz-sprawdzanie : $\mapsto$ DenTrn

włącz-sprawdzanie().sta =

jest-błąd.sta $\rightarrow$ sta

niech

(śro, (wrt, 'OK')) = sta

wrt.sprawdź = 'tak' $\rightarrow$ 'sprawdzanie-jest-włączone'

prawda $\rightarrow$ (śro, (wrt[sprawdź/'tak'], 'OK'))

Podobnie jak w przypadku instrukcji również dla transakcji wprowadzam konstruktor łączenia sekwencyjnego:

sekwencja-trn : DenTrn x DenTrn $\mapsto$ DenTrn

sekwencja-trn.(dtr-1, dtr-2) = dtr-1 • dtr-2

Ostatni konstruktor odnoszący się do transakcji tworzy z transakcji instrukcję. Dla jego zdefiniowania trzeba wprowadzić pomocniczą funkcję usuwającą wszystkie kopie bezpieczeństwa. Jej definicja jest oczywiście rekurencyjna:

usuń-wszystkie-kopie-bezp : $\mapsto$ DenTrn
 usuń-wszystkie-kopie-bezp().sta =
 jest-błąd.sta $\rightarrow$ sta
niech
 ((śrt, śrp), (wrt, 'OK')) = sta
 grp = śrt.graf-rp
 grp = $\emptyset$ $\rightarrow$ sta
 {ide} = grp $\rightarrow$ usuń-kopię-bezp.ide.sta
 {ide-1,...,ide-n} = grp $\rightarrow$ usuń-kopię-bezp.ide • usuń-wszystkie-kopie-bezp.()

Konstruktor tworzenia instrukcji z transakcji definiuję w sposób następujący:

trn-na-ins : DenTrn $\mapsto$ DenIns
 trn-na-ins.dtr.sta =
 jest-błąd.sta $\rightarrow$ sta
prawda $\rightarrow$ [usuń-wszystkie-kopie-bezp.() • włącz-sprawdzanie.()]. sta

Ten konstruktor służy do transformacji bloku transakcji (być może jednoelementowego) w instrukcję. Przy opuszczaniu bloku następuje usunięcie wszystkich kopii bezpieczeństwa i włączenie sprawdzenia.

Jak więc widzimy, mechanizm transakcji służy w zasadzie do wykonywania takich sekwencji przekształceń tabel, które pozwalają na chwilowe wyłączanie opcji sprawdzania spójności bazy, a także na tworzenie kopii bezpieczeństwa i korzystanie z nich dla przywracania początkowego stanu bazy.

12.7.6.5 Tabelowe instrukcje globalne

Ogólnie przez *tabelowe instrukcje globalne* rozumiem instrukcje, które poza modyfikacją wskazanej tabeli dokonują modyfikacji innych tabel w bazie w taki sposób, aby nie zostały naruszone więzy spójności. Na przykład w standardzie SQL są to instrukcje usuwanie wiersza w trybie CASCADE (rozdział 11.5), co może spowodować kaskadowe usunięcie z tabeli podrzędnej wszystkich wierszy wskazujących na wiersz usuwany z tabeli nadrzędnej. W tym przypadku „kaskadowo” oznacza, że jeżeli tabela podrzędna jest nadrzędną dla innej tabeli, to z tej innej należy również usunąć odpowiednie wiersze, co może spowodować konieczność usunięcia wierszy z kolejnych tabel itd. Schemat definicji takiego konstruktora pokazuję poniżej:

usuń-wi-kas : Identyfikator x Transfer $\mapsto$ DenIns
 usuń-wi-kas.(ide, tra).sta =
 jest-błąd.sta $\rightarrow$ sta
 wrt.ide = ? $\rightarrow$ sta $\leftarrow$ 'niezadeklarowany-identyfikator'

niech

$$(\text{śro}, (\text{wrt}, 'OK')) = \text{sta}$$

$$(\text{kom-t}, \text{jar}) = \text{wrt.ide}$$

$$\text{rodzaj.kom-t} \neq 'Tq' \quad \rightarrow 'oczekiwana-tabela'$$
niech

$$\text{kom-n} = \text{Km}[\text{usuń-wi-z-tab}].(\text{kom-t}, \text{tra})$$

$$\text{kom-n} : \text{Błąd} \quad \rightarrow \text{sta} \leftarrow \text{kom-n}$$
niech

$$\text{sta-n} = (\text{śro}, (\text{wrt}[\text{ide}/(\text{kom-n}, \text{jar})], 'OK'))$$

$$\text{naruszone-ja}(\text{kom-n}, \text{jar}, \text{sta-n}) \quad \rightarrow \text{sta} \leftarrow 'naruszone-jarzmo-tabelowe'$$

$$\text{naruszona-rp}(\text{kom-n}, \text{sta-n}) \quad \rightarrow \text{usuń-niespójności.sta-n}$$

$$\text{prawda} \quad \rightarrow \text{stan-n}$$

Ta instrukcje powoduje usunięcie wiersza z tabeli za pomocą konstruktora kompozytów $\text{Km}[\text{usuń-wi-z-tab}]$, a następnie, o ile nie zostało naruszone jarzmo tabelowe, ale została naruszona relacja podrzędności, to uruchamia procedurę usuń-niespójności . Tej procedury nie definiuję explicite traktując ją jako parametr modelu. Jej definicja wymagałaby dłuższych rozważań dotyczących algorytmów przeszukiwani grafów nadrzędności, których dyskusja wykraczałaby poza zakres niniejszej książki.

12.7.6.6 Tabelowe instrukcje lokalne

Tym mianem opatruję instrukcje zmieniające jedynie tabelę, której dotyczą. One albo tworzą nową tabelę albo zmieniają istniejącą odwołując się uniwersalnego konstruktora przypisań dla tabel (rozdział 12.7.6.3) i do konstruktorów denotacji wyrażeń tabelowych (rozdział 12.7.2). W zasadzie można by tych instrukcji nie wprowadzać na poziomie modelu udostępniając w języku przypisanie tabelowe. Ponieważ jednak takiego przypisania w standardzie SQL nie ma (co nie oznacza, że nie może go być w **Lingua-SQL**) podaję poniżej kilka przykładów konstruktorów denotacji tabelowych instrukcji lokalnych.

$$\text{dołoż-wi} : \text{DenWyrDan} \times \text{Identyfikator} \mapsto \text{DenIns}$$

$$\text{dołoż-wi}(\text{dwd}, \text{ide},) =$$

$$\text{przypisz-tb}(\text{ide}, \text{Kdd}[\text{Km}[\text{dołoż-wi-do-tb}]].(\text{dwd}, \text{ide}))$$

$$\text{połącz} : \text{Identyfikator} \times \text{Identyfikator} \times \text{Identyfikator} \mapsto \text{DenIns}$$

$$\text{połącz}(\text{ide-n}, \text{ide-1}, \text{ide-2}) = \quad (n - \text{nowa tabela})$$

$$\text{przypisz-tb}(\text{ide-n}, \text{Kdd}[\text{Km}[\text{połącz-tb}]].(\text{zmienna-dan.ide-1}, \text{zmienna-dan.ide-2}))$$

$$\text{przetnij} : \text{Identyfikator} \times \text{Identyfikator} \times \text{Identyfikator} \mapsto \text{DenIns}$$

$$\text{przetnij}(\text{ide-n}, \text{ide-1}, \text{ide-2}) =$$

przypisz-tb.(ide-n, Kdd[Km[przetnij-tb]].(zmienna-dan.ide-1, zmienna-dan.ide-2))

twórz-ref: Identyfikator x Identyfikator x Identyfikator x Identyfikator

x Transfer $\mapsto$ DenIns

twórz-ref.(ide-n, ide-t1, ide-t2, ide-k, tra) = (k – kolumna)

przypisz-tb.(ide-n, Kdd[twórz-tb-ref].(zmienna-dan.ide-t1, zmienna-dan.ide-t2, ide-k, tra))

zmień-ko : Identyfikator x KompozytB x Transfer x Transfer $\mapsto$ KompozytB

zmień-ko.(ide, kom, tra, jar) =

przypisz-tb.(ide, Kdd[Km[zmień-ko-w-tb]].(ide, kom, tra, jar))

Zauważmy, że w pierwszym i ostatnim z tych konstruktorów identyfikator *ide* występuje dwukrotnie, zarówno jako argument przypisania, jak i jako argument konstruktora kompozytu tabelowego. Oznacza to, że każdy z nich będzie używany wyłącznie do modyfikacji tabeli, ale nie do tworzenia nowej tabeli. To oczywiście decyzja inżynierska związana ze standardem SQL.

12.7.6.7 Zapytania

Zapytania są bardzo podobne do instrukcji prostych, z tym, że w wyniku ich działania powstaje zawsze nowa tabela o przypisana identyfikatorowi systemowemu *monitor*. W związku z tym stosuje się do nich uproszczone przypisania *przypisz-mo*, które nie narusza żadnych więzów, bo transfer powstającej wartości jest *PP*.

12.7.6.8 Instrukcja wymiany transferu

Definicja konstruktora z tej grupy, opisanego w rozdziale 6.1.5,

wymień-tr : Identyfikator x DenWyrTra $\mapsto$ DenIns,

przenosi się na przypadek SQL-owy bez żadnych zmian. Oczywiście rozbudowaniu podlega dziedzina denotacji wyrażeń transferowych.

12.7.6.9 Kursory

Kursorowy (rozdział 11.10) to mechanizmy pozwalające na pobieranie z tabeli wiersza za wierszem. Na gruncie naszego modelu można je łatwo wprowadzić, np. przez dodanie do tabeli dodatkowej kolumny numerującej wiersze kolejnymi liczbami naturalnymi.

12.7.6.10 Perspektywy

Perspektywy to w zasadzie procedury odwołujące się do instrukcji tabelowych. Można je zdefiniować jako wydzielone instrukcje lub udostępnić w języku możliwość deklarowania procedur operujących na tabelach.

12.7.6.11 Instrukcje bazodanowe

Przyjmę, że w **Lingua-SQL** wartościowanie początkowe wykonania programu może nieść pewną liczbę zmiennych z przypisanymi wartościami bazodanowymi. To oczywiście

uproszczenie w stosunku do opisanego w rozdziale 10 modelu obiektowego, którego pełne wykorzystanie w tym miejscu pozostawiam czytelnikowi.

Dodatkowo założę, że identyfikatorom systemowym w stanie inicjalnym są zawsze przypisane niżej określone wartości domyślne:

$\text{śrt.graf-rp} = \emptyset$
 $\text{wrt.kopie} = \emptyset,$
 $\text{wrt.monitor} = \Omega$ (interpretujemy jako brak danych do wyświetlenia)
 $\text{wrt.sprawdź} = \text{'tak'}$

Przy tych założeniach, każdy program bazodanowy w **Lingua-SQL**, który ma operować na jakichś tabelach, musi albo utworzyć tabele własne — z których później powstanie baza — albo rozpocząć od zagnieżdżenia w pamięci jednej z istniejących już baz. W **Lingua-SQL** będziemy więc mieli jedynie dwie instrukcje bazodanowe operujące tabelami, a ponadto dwie instrukcje modyfikujące graf podrzędności. Ich konstruktory definiuję poniżej przyjmując kolejny raz pewne techniczne uproszczenia, by nie zamulać definicji nadmiarem szczegółów.

Aktywizowanie bazy

aktywizuj : Identyfikator $\mapsto$ DenIns

aktywizuj.ide.sta =

jest-błąd.sta $\rightarrow$ sta

niech

$((\text{śrt}, \text{śrp}), (\text{wrt}, \text{'OK'})) = \text{sta}$

$\text{śrt.graf-rp} = !$ $\rightarrow$ sta $\blacktriangleleft$ 'baza-aktywna-już-istnieje'

$\text{wrt.ide} = ?$ $\rightarrow$ sta $\blacktriangleleft$ 'zmienna-nieznana'

nie $\text{wrt.ide} : \text{WarBda}$ $\rightarrow$ sta $\blacktriangleleft$ 'oczekiwana-baza'

niech

$(\text{rbd}, \text{grp}) = \text{wrt.ide}$

prawda $\rightarrow ((\text{śrt}[\text{graf-rp}/\text{grp}], \text{śrp}), (\text{wrt} \blacklozenge \text{rbd}, \text{'OK'}))$

Ta instrukcja przesłania aktualne wartościowanie rekordem aktywizowanej bazy, co oznacza, że umieszcza w nim identyfikatory tabel przypisane wartościom tabelowym, a typologicznej zmiennej systemowej *graf-rp* przypisuje graf podrzędności tejże bazy. Oczywiście wcześniej wykonuje odpowiednie sprawdzenia. Instrukcja nie pozwala na aktywizowanie dwóch baz jednocześnie (decyzja inżynierska). Przypominam, że *WarBda* jest dziedziną wartości bazodanowych zdefiniowaną w rozdziale 12.6.

Pozostała instrukcja bazodanowa zapisuje wszystkie aktualne wartości tabelowe w bazie o zadanej nazwie, a wartościowanie czyści ze wszystkich wartości innych niż bazowe.

archiwizuj : Identyfikator $\mapsto$ DenIns

archiwizuj.ide.sta =

jest-błąd.sta → sta

niech

(śro, (wrt, 'OK')) = sta

śrt.graf-rp = ? → sta ◀ 'brak-bazy-do-zapisania'

wrt.ide = ! → sta ◀ 'identyfikator-zajęty'

rbd = tylko-tabele.wrt

nowe-wrt = wyczyść-z-niebazowych.wrt

wbd = (rbd, śrt.graf-rp)

prawda → (śro, (nowe-wrt[ide/wbd], 'OK'))

W tej definicji użyłem dwóch funkcji pomocniczych — tylko-tabele oraz wyczyść-z-niebazowych — których oczywiste definicje pomijam. Przyjąłem też, że instrukcja nie pozwala na nadpisanie wcześniej zapisanej bazy nową bazą. To oczywiście decyzja inżynierska, a nie konieczność.

W tej definicji można też zapisać zasadę, że tabele, które uznamy za „chwilowe”, np. niektóre perspektywy, nie podlegają archiwizowaniu. W tym celu można np. założyć, że ich identyfikatory są jakoś specjalnie oznaczane.

Zauważmy też, że archiwizacja bazy, która przypisuje bazę do wskazanego identyfikatora, nie wymaga, aby ten identyfikator został wcześniej zadeklarowany.

Konstruktory generujące instrukcje modyfikujące graf podrzędności mamy dwa, odpowiadają dodaniu i usunięciu krawędzi grafu.

deklaruj-podrzędność : Identyfikator x Identyfikator x Identyfikator → DenIns

deklaruj-podrzędność.(ide-p, ide, ide-n).sta =

jest-błąd.sta → sta

niech

((śrt, śrp) (wrt, 'OK')) = sta

wrt.ide-i = ? → 'brak-takiej-tabeli' dla i = p, n

niech

(kom-i, tra-i) = wrt.ide-i dla i = p, n

rodzaj.kom-i ≠ 'Tq' → 'oczekiwana-tabela' dla i = p, n

niech

((tab-i, ('Tq', wie-d-i, ('Wq', rwi-i)), jar-i) = wrt.ide-i dla i = p, n

rko-i.ide = ? → 'brak-takiej-kolumny' dla i = p, n

niech

grp = śrt.graf-rp (przypominam, że grp może być pusty)

(ide-p, ide, ide-n) : grp → 'deklaracja-redundantna'

kom-p Pod[ide] kom-n $\rightarrow$ grp | {(ide-p, ide, ide-n)}
prawda $\rightarrow$ 'relacja-podrzędności-niespełniona'

Ta instrukcja sprawdza przed dodaniem nowej krawędzi grafu podrzędności, czy wyznaczona przez niego relacja rzeczywiście zachodzi. Jeżeli tabele, których dotyczy są duże, może to być obliczeniowo dość kosztowne, jednak nie daje się tego uniknąć, jeżeli chcemy zachować zasadę, że bazodanowe instrukcje nie naruszają spójności bazy.

Druga operacja nie wymaga takiego sprawdzenia, gdyż jedynie usuwa krawędź z grafu podrzędności.

odwołaj-podrzędność : Identyfikator x Identyfikator x Identyfikator $\mapsto$ DenIns

odwołaj-podrzędność.(ide-p, ide, ide-n).sta =

jest-błąd.sta $\rightarrow$ sta

niech

((śrt, śrp), (wrt, 'OK')) = sta

grp = śrt.graf-rp

(ide-p, ide, ide-n) : grp $\rightarrow$ grp – {(ide-p, ide, ide-n)}

prawda $\rightarrow$ 'brak-takiej-podrzędności'

12.8 Składnia konkretna

Dla czytelnika, który dotarł do tego rozdziału zaprojektowanie składni konkretnej obejmującej QSL-owe mechanizmy **Lingua-SQL** powinno być stosunkowo proste. Ograniczę się więc do nowych reguł związanych z SQL. Opisana niżej składnia nie jest być może optymalna, gdyż zawiera dość długie słowa kluczowe, jednak — jak już nieraz w tej książce — chodzi mi raczej o pokazanie metody niż o zbudowanie konkretnego języka. Z tego samego powodu nie jest też szczególnie bliska składni standardu SQL. Długie słowa kluczowe będące wolnym tłumaczeniem na j. angielski nazw konstruktorów denotacji mają ułatwić czytelnikowi powiązanie klauzul gramatycznych z tymiż konstruktorami. Zauważmy, że w stosunku do **Lingua-3** doszła nowa kategoria składniowa obejmująca transakcje. Słowa kluczowe **ed**, **et** i **ei** czytamy odpowiednio jako „end of declaration”, „end of transaction” oraz „end of instruction”.

Wyrażenia danologiczne

wyd : WyrDan =

...

(tu stoją wszystkie klauzule z **Lingua-3**)

Wyrażenia generujące kompozyty puste

empty-bool

|

empty-number

|

empty-word

|

...

Wyrażenia wierszowe

```

row Identyfikator val WyrDan ee |
expand-row WyrDan at Identyfikator by WyrDan ee |
reduce-row WyrDan at Identyfikator ee |
row WyrDan at Identyfikator ee |
change-row WyrDan at Identyfikator by WyrDan ee |

```

Wierszowe wyrażenia tabelowe

```

table WyrDan at Identyfikator ee |
add-row WyrDan to Identyfikator ee |
delete-row WyrTra from Identyfikator ee |
remove Identyfikator from Identyfikator ee |
clear Identyfikator with WyrTra ee |
intersect Identyfikator with Identyfikator ee |
union Identyfikator with Identyfikator ee |

```

Kolumnowe wyrażenia tabelowe

```

add-column Identyfikator with WyrDan to Identyfikator ee |
remove-column Identyfikator from Identyfikator ee |
filter-columns ParAktualne from Identyfikator ee |
remove-column Identyfikator from Identyfikator ee |
update-column Identyfikator in Identyfikator with WyrTra
                                where WyrTra ee |

```

Wyrażenia tworzące tabelę pochodną

```

table Identyfikator with Identyfikator at Identyfikator
                                where WyrTra ee |

```

Wyrażenia transferowe

wtr : WyrTra =

```

... (tu stoją wszystkie klauzule z Lingua-3)
row . Identyfikator |
unique |
all WyrTra ee

```

Wyrażenia typologiczne

wyt :WyrTyp =

```

... (tu stoją wszystkie klauzule z Lingua-3)
row-type Identyfikator as WyrTyp ee |
expand-row-type WyrTyp by Identyfikator as WyrTyp ee |
table-type WyrDan as WyrTra ee

```

Definicje stałych typologicznych

W tej grupie nie ma nowych klauzul. Oczywiście „dawne” klauzule odwołują się do nowych wyrażeń typologicznych

Deklaracje zmiennych danologicznych

dez : DekZmi =

```

... (tu stoją obie klauzule z Lingua-3)
create table Identyfikator as WyrTyp ed

```

Transakcje

trn : Transakcja =

```

add WyrDan to Identyfikator et |
delete WyrDan from Identyfikator et |
exclude Identyfikator from Identyfikator et |
add column Identyfikator with WyrDan to Identyfikator et |
drop column Identyfikator from Identyfikator et |
select columns ParAktualne from Identyfikator et |
update Identyfikator at Identyfikator with WyrTra et |
savepoint Identyfikator et |
release savepoint Identyfikator et |
rollback Identyfikator et |
rollback Identyfikator if WyrDan et |
constraints off |
constraints on |
Transakcja ; Transakcja

```

Instrukcje

ins : Instrukcja =

...

(tu stoją wszystkie klauzule z **Lingua-3**)**Instrukcje tabelowe**

```

delete cascade WyrTra from Identyfikator ei |
add row WyrDan to Identyfikator ei |
union Identyfikator with Identyfikator into Identyfikator ei |
intersect Identyfikator with Identyfikator into Identyfikator ei |
create Identyfikator from Identyfikator and Identyfikator col Identyfikator
                                where WyrTra ei |
modify column Identyfikator in Identyfikator by WyrTra
                                where WyrTra ei |

```

Instrukcje bazodanowe

```

activate Identyfikator |
archive as Identyfikator |
set reference of Identyfikator et Identyfikator to Identyfikator ei |
clear reference of Identyfikator et Identyfikator to Identyfikator ei |

```

Zapytania

Zapytania to przypisania, a więc instrukcje, które tworzoną tabelę przypisują identyfikatorowi systemowemu **monitor** i nie dokonują żadnych sprawdzeń, bo z tym identyfikatorem nie jest związany żaden typ tabelowy. Ich denotacje są więc nieco odmienne od denotacji odpowiadających im instrukcji, co powoduje, że ich składnie muszą się też odpowiednio różnić. Przyjmę, że powstają one ze składni odpowiadających im instrukcji przez dodanie przedrostka **show**.

que : Query =

```

show Identyfikator |
show union Identyfikator with Identyfikator into Identyfikator ei |
show intersect Identyfikator with Identyfikator into Identyfikator ei |
show create Identyfikator from Identyfikator and Identyfikator
                                col Identyfikator where WyrTra ei

```

Na koniec jeszcze ważna uwaga metodologiczna. Otóż w **Lingua-SQL** dopuściłem na poziomie składni pełen asortyment wyrażeń odpowiadających konstruktorom denotacji wyrażeń daniologicznych. Dotyczy to w szczególności wyrażeń tabelowych. Dopuściłem też przypisania, w których takie wyrażenia mogą występować. Takie środki językowe są jednak dość odległe od standardu SQL, a także mogą prowadzić — przy złożonych wyrażeniach — do mało czytelnych programów oraz trudnych do sformułowania reguł konstruowania programów poprawnych.

Alternatywą do przyjętego rozwiązania może być dopuszczenie w **Lingua-SQL** jedynie wyrażeń z **Lingua-3** oraz wyrażeń wierszowych, natomiast wyzbycie się wyrażeń tabelowych na rzecz instrukcji tabelowych pozwalających na budowanie tabel krok za krokiem. Nie oznacza

to, że na poziomie modelu nie możemy wprowadzić konstruktorów denotacji wyrażeń danologicznych. Jak widzieliśmy, takie konstruktory są przydatne przy wstępującym definiowaniu konstruktorów denotacji instrukcji. Jednakże przystępując do zdefiniowania składni języka możemy podjąć inżynierską decyzję, że pewnych konstruktorów denotacji nie włączamy do sygnatury algebry traktując je jako funkcje pomocnicze. Taka decyzja spowoduje, że ich odpowiedniki nie pojawiają się też na poziomie składni.

12.9 Składnia kolokwialna

Wydaje się, że większość z nowych konstrukcji składniowych **Lingua-SQL** nie wymaga wprowadzania kolokwializmów. Pewnie wystarczy uczynić je bardziej przyjazne dla użytkownika na poziomie składni konkretnej. Natomiast może być zasadne wprowadzenie kolokwializmu dla deklaracji typów tabelowych, który byłby bliski typowej składni języków SQL-owych. Rozważmy więc SQL-owe przykład deklaracji zmiennej tabelowej:

```
create table Pracownicy with
  Nazwisko      Varchar(20)    NOT NULL,
  Stanowisko    Varchar(9),
  Pensja        Number(5)      DEFAULT 0,
  Premia        Number(4)      DEFAULT 0,
  Id_działu     Number(3)      REFERENCES Działy,
  CHECK (Premia < Pensja)
ed
```

Transformacja przywracająca zamieniałaby tę deklarację na sekwencję składającą się z deklaracji zmiennej tabelowej, po której następuje instrukcja bazodanowa:

```
create table Pracownicy as
  table-type wyr_dan with wyr_tra ee
ed;
set reference of Pracownicy et Id_działu to Działy ei
```

gdzie `wyr_dan` oraz `wyr_tra` reprezentują wyrażenie danologiczne i odpowiednio wyrażenie transferowe.

Rozwijając wyrażenie danologiczne konstruktorami tworzenia i rozszerzania wiersza, a wyrażenie transferowe — konstruktorami wyrażeń transferowych, otrzymujemy poniższą wersję konkretną naszej deklaracji kolokwialnej:

create table Pracownicy as	<i>początek deklaracji zmiennej tablicowej</i>
table-type	<i>początek wyrażenia typologicznego</i>
expand-row	<i>początek wyrażenia danologicznego</i>
expand-row	
expand-row	
expand-row	
row Nazwisko val empty-word ee	

```

        by Stanowisko val empty-word ee
    by Pensja val 0 ee
    by Premia val 0 ee
    by Id_działu by empty-number ee      koniec wyrażenia danologicznego
with                                     początek wyrażenia transferowego (jarzmowego)
    all
        varchar(20) (row.Nazwisko)      and
        not-null (row.Nazwisko)          and
        varchar(9) (row.Stanowisko)      and
        number(5) (row.Pensja)           and
        number(4) (row.Premia)           and
        number(3) (row.Id_działu)        and
        row.Premia < row.Pensja
    ee                                    koniec wyrażenia transferowego (jarzmowego)
    ee                                    koniec wyrażenia typologicznego
ed ;                                    koniec deklaracji zmiennej tablicowej
set reference of Pracownicy et Id_działu to Działy ei

```

Oczywiście `varchar(20)`, `varchar(9)`,... są nazwami składniowymi odpowiednio zdefiniowanych predykatów. Należy jeszcze raz podkreślić, że w tym przykładzie jedna „jednostka składniowa” z poziomu kolokwialnego (deklaracja) zostaje przez transformację przywracająca zmieniona na sekwencję deklaracji i instrukcji.

12.10 Reguły konstruowania programów poprawnych

Wzbogacenie wcześniejszych wersji **Lingua** do **Lingua-SQL** polegało przede wszystkim na istotnym rozbudowaniu algebr danych i typów, natomiast nowe instrukcje, to przede wszystkim instrukcje modyfikacji tabel odwołujące się na poziomie denotacyjnym do uogólnionego przypisania. Dla twórcy reguł walidacyjnych oznacza to, że w zasadzie trzeba pomyśleć jedynie o asortymencie nowych warunków i właściwości (rozdziały 8.2 i 8.4.1). Nad tym jednak warto się zastanowić dopiero wtedy, gdy zostanie ustalona jakaś praktyczna wersja **Lingua-SQL**.

13 Co zostało jeszcze do zrobienia

Mimo że książka jest już dość spora, większość tematów została w niej jedynie naszkicowana. To co jest jeszcze do zrobienia może starczyć na kilka kolejnych książek, a także jako tematyka badawcza i wdrożeniowa dla wielu osób i zespołów. Poniżej wstępna lista zadań do wykonania, która z pewnością nie jest kompletna. Obejmuje zarówno zadania teoretyczne jak i programistyczne (implementacyjne).

13.1 Podstawy

13.1.1 Rozbudowanie modelu dla Lingua

Języki z serii **Lingua**, być może poza **Lingua-3** (programowanie obiektowe), obejmują w zasadzie dość tradycyjne narzędzia programistyczne powstałe w większości w latach 1960-1980. Ponieważ są one dziś obecne w ogromnej większości języków programowania, należało od nich zacząć, co nie oznacza jednak, że należy na nich poprzestać. W mojej ocenie kolejny krok to rozbudowa naszego modelu o narzędzia bardziej współczesne, jak. np. języki skryptowe typu HTML, czy też programowanie współbieżne oparte na modelach Mazurkiewicza i/lub Petriego.

Kilka mniejszych zadań badawczych zostało też wskazanych w podstawowej części książki.

13.1.2 Uzupełnienie modelu dla Lingua

Opracowanie pełnych modeli denotacyjnych dla praktycznych wersji kolejnych języków z serii **Lingua**. Te modele powinny obejmować nie tylko denotacje, składnię i semantykę, ale też reguły konstruowania programów poprawnych. Należałoby się też przyjrzeć pojęciu asercji i dekretowaniu obszarów obowiązywania warunków, które w rozdziale 8.3 zostały w zasadzie jedynie naszkicowane.

13.1.3 Zasady pisania podręczników użytkownika

Denotacyjne modele języków programowania stwarzają okazję do zrewidowania aktualnych praktyk widocznych w podręcznikach języków programowania. Nowe praktyki powinny z jednej strony opierać się na modelach denotacyjnych, ale z drugiej — nie zakładać, że użytkownik będzie musiał zbyt wiele o tych modelach wiedzieć. Użytkownikowi języka należy dostarczyć jedynie tyle wiedzy teoretycznej, ile jest konieczne do zrozumienia praktycznych cech języka. Jednocześnie jednak powinny to być podręczniki dla profesjonalnych programistów, a nie — jak niestety nierzadko — dla amatorów zabierających się za programowanie bez żadnej wiedzy podstawowej. Oczywiście książki wprowadzające do zawodu programisty są również potrzebne, one jednak powinny wyjaśniać zasady budowania programów bez wchodzenia w szczegóły techniczne konkretnych języków.

13.2 Implementacje

W tym obszarze proponowałbym, aby jedynie dla (stosownie uzupełnionego) **Lingua-1** implementację napisać w którymś z istniejących języków, np. w Phytonie, a następnie implementacje kolejnych warstw naszego języka, a także narzędzia dla projektantów języków i programistów, tworzyć z wykorzystaniem wcześniej zbudowanych warstw **Lingua**.

13.2.1 Narzędzia dla projektantów języków

1. System generujący gramatykę składni abstrakcyjnej języka z sygnatury (meta-opisu) algebry denotacji.
2. System wspomagający przekształcanie gramatyki składni abstrakcyjnej w gramatykę składni konkretnej.
3. System wspomagający generowanie aplikacji przywracającej ze składni kolokwialnej do konkretnej.
4. Edytor wspomagający pisanie definicji konstruktorów denotacji. Ten edytor powinien umożliwiać nie tylko przyjazne dla czytelnika formatowanie definicji, ale przede wszystkim dbać o to, aby nie pojawiała się redundancja, tj. sytuacja, gdy jedno oznaczenie zostało zdefiniowane na więcej niż jeden sposób.
5. Generator klauzul semantycznych przyjmujący jako źródła gramatykę konkretną i definicję konstruktorów.
6. Generator kodu interpretera/kompilatora z klauzul semantycznych.

13.2.2 Narzędzia dla programistów w Lingua

System wspomagający pisanie programów przy użyciu reguł budowania metaprogramów poprawnych.

13.2.3 Podręczniki

Aby metodologia zawarta w **Lingua** mogła nabrać praktycznego znaczenia, muszą powstać odwołujące się do tej metodologii podręczniki programisty. Oczywiście powinny się one opierać na zasadach, o których mowa w rozdziale 13.1.3. Prawdę mówiąc oba nurty powinny być rozwijane równolegle. Aby określić ogólne zasady pisania podręczników trzeba zacząć eksperymentować z takim podręcznikiem, a jednocześnie te eksperymenty powinny korzystać z zasad, które dało się wypracować na poziomie ogólnym.

13.3 Eksperymenty programistyczne

Aby nasza idea budowania programów poprawnych została zauważona przez środowisko praktyków IT, trzeba pokazać przekonujące przykłady zastosowań. Myślę, że na początek można by się zająć się mikroprogramami, a to z następujących powodów:

1. są to programy o stosunkowo niewielkiej liczbie linii kodu,
2. problem poprawności jest dla nich szczególnie krytyczny
3. krytyczny jest też problem optymalizacji czasu wykonania i rozmiaru używanej pamięci.

Oczywiście każdy eksperymentalnie napisany przykład takiego programu należałoby poddać powszechnie używanym testom przemysłowym.

13.4 Budowania społeczności zwolenników **Lingua**

Przedstawione w niniejszej książce metody budowania języków programowania i tworzenia programów mogą spotkać się z różnymi ocenami, jednakże jedno jest chyba pewne, a mianowicie że odbiegają one dość daleko od dzisiejszej praktyki. To co proponuje książka, to dość daleko idąca zmiana, a każda poważna zmiana rodzi zarówno przeciwników jak i zwolenników. Przeciwników należy przekonać, a zwolenników trzeba zdobyć. I oczywiście trzeba zacząć od drugiego z tych zadań.

Aby to zadanie zrealizować, trzeba albo potencjalnym zwolennikom dać do ręki choćby najprostsza, ale praktyczną, wersję **Lingua**, albo też zachęcić ich do zbudowania własnej. Pierwsza droga wydaje się mało prawdopodobna, bo wymagałaby znalezienia inwestora, który sfinansowałby budowę prototypu czegoś dziwnego i nieznanego, na co raczej liczyć nie można. Pozostaje więc druga droga, co oznacza budowę **Lingua** przez ochotników dla ochotników, a więc założenie, że będzie to produkt bezpłatny i ogólnie dostępny, jak choćby Linux, Joomla! czy Drupal. Jednakże „ogólnie dostępny” nie może w tym przypadku oznaczać powszechnej dostępności do jego tworzenia (open source), bo to mogłoby prowadzić do matematycznie niepoprawnych konstrukcji i w rezultacie do niepoprawnych reguł budowania programów.

Społeczność budowniczych **Lingua** powinna więc wypracować reguły przyjmowania nowych jej członków i udzielania im prawa do włączenia się w tok prac implementacyjnych. Ta zasada nie oznacza oczywiście, że należałoby jakkolwiek ograniczać badania naukowe w zakresie naszej metody, te bowiem z istoty rzeczy podlegają naturalnej ocenie przez środowisko akademickie.

14 ANEKS 1 — Drzewa uogólnione

Ten aneks powstał dość dawno, wymaga więc przejrzenia pod kątem zgodności z pozostałą częścią książki.

Drzewa uogólnione, jak je będę nazywał w niniejszej książce, to drzewa, w których węzłach mogą stać dowolne dane znane nam z **Lingua-A** (rozdział 5), a w tym dane strukturalne, a także drzewa. Dyskusję tak rozumianych drzew wyłączyłem z rozdziału 5, by nie obciążać zbyt wiele szczegółami technicznymi głównego nurtu mojego wykładu. W niniejszym aneksie przedstawię rozszerzenie **Lingua-A** o drzewa, co oznacza rozszerzenie wszystkich składających się na model tego języka algebr. Rozszerzony język nazwę **Lingua-AD**. W naturalny sposób będzie można zbudować nad nim kolejne warstwy języka imperatywnego.

14.1 Nośniki wyjściowe algebry danych

Nośniki wyjściowe do zbudowania dwurodzajowej rozszerzonej algebry danych są następujące:

ide	:	Identyfikator	
boo	:	Boolowska	$= \{pp, ff\}$
lic	:	Liczba	
sło	:	Słowo	$= \{ \} \text{Alfabet}^c \{ \}$
lis	:	Lista	$= \text{Dana}^{c+}$
tab	:	Tablica	$= \text{Liczba} \Rightarrow \text{Dana}$
rek	:	Rekord	$= \text{Identyfikator} \Rightarrow \text{Dana}$
ojc	:	Ojciec	$= \text{Dana}$
syn	:	Syn	$= \text{Dana}$
drz	:	Drzewo	$= \text{Ojciec} \times \text{Syn}^{c+}$
dan	:	Dana	$= \text{Boolowska} \mid \text{Liczba} \mid \text{Słowo} \mid \text{Lista} \mid \text{Tablica} \mid \text{Rekord} \mid \text{Drzewo}$

Jak widzimy, mamy tu nie tylko nowe nośniki, ale też nowe elementy w nośnikach list, tablic i rekordów, gdyż powiększony został nośnik danych.

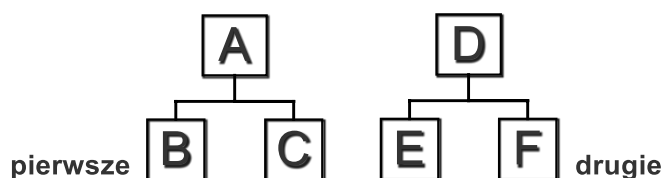
Listy tablice i rekordy zostały zdefiniowane w nowej algebrze w tradycyjny sposób, o ile pominiemy fakt, że ich elementami mogą być dowolne dane, w tym strukturalne. Jednakże matematykowi łatwo jest wyobrazić sobie kombinacje takich struktur, np. listy rekordów przechowujących tablice. Z drzewami sytuacja jest nieco bardziej złożona, gdy uświadomimy sobie, że zarówno ojcem drzewa, jak i dowolnym z jego synów, może być nie tylko lista, tablica lub

rekord, ale też drzewo. Ta ostatnia sytuacja wymaga pewnego wyjaśnienia, nie występuje bowiem w tradycyjnie rozumianej teorii grafów¹²⁹.

Rozpocznijmy od prostego przykładu dwóch drzew pokazanych na Rys. 14.1-1. W naszej notacji te drzewa odpowiadają dwóm krotkom:

pierwsze = (A, (B, C))

drugie = (D, (E, F))



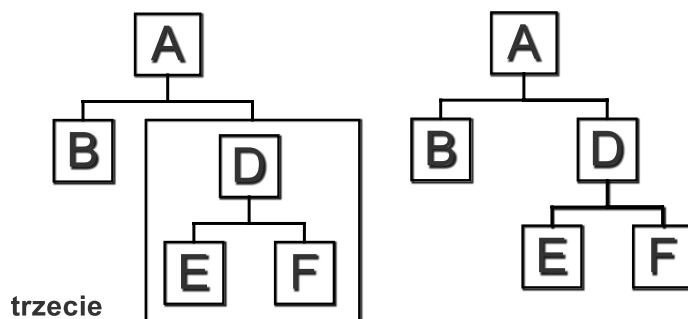
Rys. 14.1-1 Dwa drzewa proste

Zauważmy, że w tej notacji nasze przykładowe drzewa nie stanowią — jak przy teoriografowym rozumieniu drzew — grafów, których węzły są etykietowane literami od A do F, ale są to grafy, których węzłami są etykiety. Położenie węzła w grafie wyznacza położenie etykiety w krotce i na odwrót.

Jeżeli teraz w drzewie pierwsze na miejsce syna C wstawimy drzewo drugie, to otrzymamy drzewo trzecie,

trzecie = (A, (B, drugie)) = (A, (B, (D, (E, F))))

którego jednym z dwóch liści będzie drzewo drugie (Rys. 14.1-2).



Rys. 14.1-2 Multidrzewo i drzewo proste

W tradycyjnym przedstawieniu teoriografowym odpowiadałoby mu drzewo proste przedstawione po prawej stronie Rys. 14.1-2, które jednak nie ma w naszym języku innego odpowiednika niż drzewo trzecie. W tradycyjnym rozumianym drzewie, tj. stojącym po prawej stronie rysunku, D jest synem A, natomiast w naszym — jest ojcem syna A, którym jest drzewo drugie. To ujęcie pozwala jedną operacją

weź-syna.(2, trzecie) = (D, (E, F))

¹²⁹ Oczywiście taką sytuację mógłbym łatwo wyeliminować z modelu drzew, nie znalazłem jednak argumentu za tym, aby tak postąpić.

wyselekcjonować całe poddrzewo drzewa pierwsze (definicja tej funkcji w rozdziale 14.2). Iterując operację **weź-syna** można wyselekcjonować dowolne poddrzewo każdego drzewa, a także każdy jego liść.

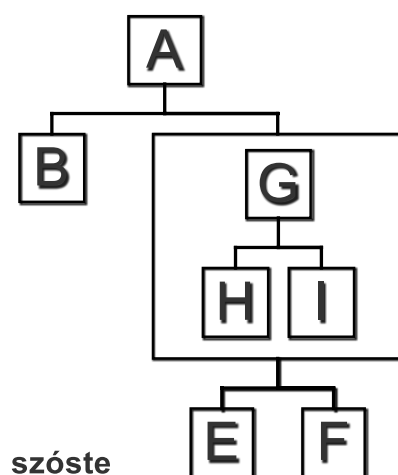
W **Lingua-AD** operujemy więc jedynie drzewami dwupiętrowymi — piętro ojca i piętro synów — ponieważ jednak ich liście mogą być również drzewami, możemy w ten sposób symulować drzewa o dowolnej liczbie pięter. Zauważmy, że analogiczną sytuację mamy z tablicami, które są zawsze tablicami jednowymiarowymi, mogącymi mieć jednak inne tablice jako swoje elementy. Z rekordami jest podobnie. Nasze tablice i rekordy pozwalają więc na zapisanie tablic i rekordów dowolnego wymiaru w sensie tradycyjnym.

Z drzewami jest jednak nieco inaczej, gdyż drzewo możemy wstawić nie tylko na miejsce dowolnego syna, ale też i na miejsce ojca. To pozwala budować struktury, których interpretacje wykraczają poza tradycyjnie rozumiane drzewa. Przyjrzyjmy się następującej konstrukcji zilustrowanej na Rys. 14.1-3. Niech:

czwarte = (G, (H, I))

piąte = (czwarte, (E, F)) = ((G, (H, I)), (E, F))

szóste = (A, (B, piąte)) = (A, (B, ((G, (H, I)), (E, F))))



Rys. 14.1-3 Drzewo niemające tradycyjnego odpowiednika

To drzewo nie odpowiada żadnemu drzewu tradycyjnemu, może mieć jednak całkiem naturalne zastosowanie, na przykład do opisu struktury kierowniczej jakiejś firmy. Taka firma miałaby prezesa A, któremu podlega wiceprezes B oraz kolegium kierownicze (G, (H, I)). To kolegium składa się z dyrektora G oraz dwóch podległych mu dyrektorów H oraz I. Kolegium zarządza (kolegialnie) dwoma podległymi pracownikami E oraz F.

14.2 Konstruktory algebry danych

Wśród konstruktorów naszej nowej algebry danych umieszczam wszystkie konstruktory z **Lingua-A**, do których dodaję trzy grupy konstruktorów związanych z drzewami:

1. tworzenie drzewa,
2. modyfikacja drzewa,

3. selekcja z drzewa.

Pamiętajmy też, że drzewa są krotkami, możemy więc stosować do nich notację używaną dla funkcji. Przypominam też, że w algebrze danych nie wprowadzamy błędów abstrakcyjnych, niektóre z jej operacji będą więc częściowe. Pierwszy konstruktor tworzy drzewo dwupoziomowe o jednym synu:

$\text{twórz-drz} : \text{Dana} \times \text{Dana} \mapsto \text{Drzewo}$

$\text{twórz-drz}(\text{dan-o}, \text{dan-s}) = (\text{dan-o}, (\text{dan-s}))$ (o – ojciec, s – syn)

Następny konstruktor pozwala na dodanie nowego syna:

$\text{dodaj-syna} : \text{Dana} \times \text{Drzewo} \mapsto \text{Drzewo}$

$\text{dodaj-syna}(\text{dan}, (\text{ojciec}, \text{krotka-synów})) = (\text{ojciec}, \text{krotka-synów} \odot (\text{dan}))$

Kolejne dwa konstruktory pozwalają na wymianę w drzewie odpowiednio ojca oraz dowolnego z jego synów:

$\text{wymień-ojca} : \text{Dana} \times \text{Drzewo} \mapsto \text{Drzewo}$

$\text{wymień-ojca}(\text{dan}, (\text{ojciec}, \text{krotka-synów})) = (\text{dan}, \text{krotka-synów})$

$\text{wymień-syna} : \text{Dana} \times \text{Liczba} \times \text{Drzewo} \rightarrow \text{Dana}$

$\text{wymień-syna}(\text{dan}, \text{lic}, (\text{ojciec}, \text{krotka-synów})) =$

$\text{krotka-synów.lic} = ? \rightarrow ?$

prawda $\rightarrow (\text{ojciec}, \text{krotka-synów}[\text{lic}/\text{dan}])$

$\text{weź-ojca} : \text{Drzewo} \mapsto \text{Dana}$

$\text{weź-ojca}(\text{ojciec}, \text{krotka-synów}) = \text{ojciec}$

$\text{weź-syna} : \text{Drzewo} \times \text{Liczba} \rightarrow \text{Dana}$

$\text{weź-syna}((\text{ojciec}, \text{krotka-synów}), \text{lic}) =$

$\text{krotka-synów.lic} = ? \rightarrow ?$

prawda $\rightarrow \text{krotka-synów.lic}$

Nad tak zbudowaną algebrą częściową o wielu nośnikach buduję teraz — analogicznie jak w rozdziale 5.2.1 — algebrę o dwóch nośnikach:

Identyfikator

Dana

której wszystkie operacje będą częściowe.

14.3 Algebra korpusów

Nowa algebra korpusów powstaje przez rozszerzenie algebry opisanej w rozdziale 5.2.2 i ma trzy nośniki:

$\text{ide} : \text{Identyfikator}$

lic : LiczbaB = Liczba | Błąd

kor : KorpusB = Korpus | Błąd

gdzie tym razem

$$\begin{aligned} \text{kor : Korpus} = & \{ \text{'boolowska'} \} \mid \{ \text{'liczba'} \} \mid \{ \text{'słowo'} \} \mid \\ & \{ \text{'L'} \} \times \text{Korpus} \mid \\ & \{ \text{'T'} \} \times \text{Korpus} \mid \\ & \{ \text{'R'} \} \times (\text{Identyfikator} \Rightarrow \text{Korpus}) \mid \\ & \{ \text{'D'} \} \times \text{Korpus} \times \text{Korpus}^{c+} \end{aligned}$$

Jak widzimy, korpusy drzew są drzewami korpusów etykietowanymi inicjałem 'D'. Wśród nośników nowej algebry znalazły się też liczby, które są potrzebne do selekcji synów z drzew.

Na nową dziedzinę korpusów rozszerzam funkcję rodzaj (rozdział 5.2.2) w oczywisty sposób:

rodzaj.('D', kor, (kor-1,...,kor-n)) = 'D'

Na korpusy drzew należy też rozszerzyć funkcje KLAN-Kr:

$$\begin{aligned} \text{KLAN-Kr.('D', kor, (kor-1,...,kor-n))} = \\ \{ (\text{dan}, (\text{dan-1},..., \text{dan-n})) \mid \text{dan : KLAN-Kr.kor oraz} \\ \text{dan-i : KLAN-Kr.kor-i dla } i = 1;n \} \end{aligned}$$

co pozwala w naturalny sposób rozszerzyć funkcję

KOR : Dana $\mapsto$ Korpus

Analogicznie jak w rozdziale 5.2.2 dla każdego konstruktora drzew definiujemy odpowiadający mu konstruktor korpusów drzewowych.

twórz-drz-kr : KorpusB x KorpusB $\mapsto$ KorpusB

$$\begin{aligned} \text{twórz-drz-kr.}(\text{kor-o}, \text{kor-s}) = & \hspace{15em} (\text{o} - \text{ojciec}, \text{s} - \text{syn}) \\ \text{kor-o : Błąd} & \hspace{5em} \rightarrow \text{kor-o} \\ \text{kor-s : Błąd} & \hspace{5em} \rightarrow \text{kor-s} \\ \text{prawda} & \hspace{5em} \rightarrow (\text{'D'}, \text{kor-o}, (\text{kor-s})) \end{aligned}$$

wymień-ojca-kr : KorpusB x KorpusB $\mapsto$ KorpusB

$$\begin{aligned} \text{wymień-ojca-kr.}(\text{kor-o}, \text{kor-d}) = & \hspace{15em} (\text{o} - \text{ojciec}, \text{d} - \text{drzewo}) \\ \text{kor-o : Błąd} & \rightarrow \text{kor-o} \\ \text{kor-d : Błąd} & \rightarrow \text{kor-d} \end{aligned}$$

niech

$$\begin{aligned} (\text{'D'}, (\text{kor}, (\text{kor-1},..., \text{kor-n}))) &= \text{kor-d} \\ \text{kor-d-nowy} &= (\text{'D'}, \text{kor-o}, (\text{kor-1},..., \text{kor-n})) \\ \text{prawda} &\rightarrow \text{kor-d-nowy} \end{aligned}$$

wymień-syna-kr : KorpusB x LiczbaB x KorpusB $\mapsto$ KorpusB

wymień-syna-t.(kor-s, lic, kor-d) = (s – syn, d – drzewo)

kor-s : Błąd $\rightarrow$ kor-s

lic : Błąd $\rightarrow$ lic

kor-d : Błąd $\rightarrow$ kor-d

niecałkowita.rze $\rightarrow$ 'oczekiwana-całkowita'

niech

('D', kor-o, (kor-1,...,kor-n)) = kor-d

krotka-typów-synów.lic = ? $\rightarrow$ 'indeks-spoza-zakresu'

niech

kor-d-nowy = ('D', kor-o, (kor-1,...,kor-n)[lic/kor-s])

prawda $\rightarrow$ kor-d-nowy

dodaj-syna-t : KorpusB x KorpusB $\mapsto$ KorpusB

dodaj-syna-t.(kor-s, kor-d) = (s – syn, d – drzewo)

kor-s : Błąd $\rightarrow$ kor-s

kor-d : Błąd $\rightarrow$ kor-d

niech

('D', kor-o, (kor-1,...,kor-n)) = kor-d

kor-d-nowy = ('D', kor-o, (kor-1,...,kor-n) © (kor-s))

prawda $\rightarrow$ kor-drz-nowy

W przypadku drzew mamy więc sytuację podobną jak w przypadku rekordów: każdy syn może być innego typu. Definicje konstruktorów **weź-ojca-kr** i **weź-syna-kr** pozostawiam czytelnikowi.

14.4 Algebra kompozytów

Nowa algebra kompozytów powstaje przez rozszerzenie algebry opisanej w rozdziale 5.2.3 polegającej na rozszerzeniu jej nośnika kompozytów przez rozszerzenie dziedziny korpusów, co oznacza, że definicja (5.2.3-1) pozostaje w mocy, tyle że z rozszerzoną dziedziną Korpus. Ta algebra ma ponownie dwa nośniki:

Identyfikator

Kompozyt

Nośnik Liczba nie jest już nam potrzebny, bo wśród kompozytów mamy kompozyty liczbowe. Konstruktory kompozytów drzewowych definiujemy zgodnie ze schematem (5.2.3-2). Trzeba oczywiście założyć, że na kompozyty drzewowe zostały odpowiednio rozszerzone funkcje **twórz-korpus** oraz **nadmiarowy**.

14.5 Algebra transferów

Algebrę transferów zdefiniowaną w rozdziale 5.2.4 rozbudowujemy w standardowy sposób o transfery selekcji ojca i syna. Można też ją rozbudować dodatkowo o jarzma opisujące właściwości drzew, np. typu wszystkie-dr lub inne.

14.6 Algebra typów

Algebra typów powstaje przez oczywiste rozbudowanie algebry typów z rozdziału 5.2.5 przez rozszerzenie jej nośników i rozbudowę listy konstruktorów. W odpowiedni też sposób są rozszerzane dziedziny Wartość oraz Stan.

14.7 Algebra denotacji wyrażeń danologicznych

Ponieważ wszystkie nowe w **Lingua-AD** konstruktory danych są transparentne wobec błędów, ich definicje wywodzą się wprost ze schematu (5.3.2-1) opisanego w rozdziale 5.3.2. W podręczniku użytkownika można je też napisać w wersji wyraźniej (rozdział 5.3.3). To kończy rozbudowę **Lingua-A** do **Lingua-AD**.

14.8 Algebra denotacji wyrażeń typologicznych

Ta algebra powstaje z algebry opisanej w rozdziale 5.3.4 przez uzupełnienie jej o nowe konstruktory odpowiadające konstruktorom typów drzewologicznych.

14.9 Algebra denotacji języka Lingua-2D

Język **Lingua-2D** budujemy nad **Lingua-AD** analogicznie jak **Lingua-2** był budowany nad **Lingua-A**. Wszystkie definicje dziedzin i konstruktorów pozostają w mocy, choć odnoszą się do rozszerzonych treści. Należy jedynie rozszerzyć definicję relacji koherentny pomiędzy korpusami przez założenie, że w przypadku korpusów drzewowych relacja ma miejsce, gdy jeden korpus powstaje z drugiego przez dodanie syna.

14.10 Programowanie walidujące w Lingua-2DV

Ogólne zasady budowania warstwy deskryptywnej dla dowolnego języka z rodziny **Lingua** zostały opisane w rozdziale 8 i tamże zilustrowane na przykładzie języka **Lingua-2V**. Wszystkie te zasady i związane z nimi konstrukcje przenoszą się bez żadnych zmian do **Lingua-2DV**. Oczywiście język można rozbudować o nowe warunki (rozdział 8.2).

15 ANEKS 2 — O podręcznikach użytkownika

15.1 O potrzebie nauczania informatyki

W dniu 3 lipca 2013 roku odbyła się w Sali Kolumnowej Pałacu Prezydenckiego konferencja inaugurująca powołanie *Szerokiego Porozumienia na Rzecz Umiejętności Cyfrowych w Polsce*, którym stała się nieformalna grupa instytucji, organizacji i firm gotowych wspierać rozwój kompetencji cyfrowych Polaków. Inicjatorami Porozumienia byli Minister Cyfryzacji i Administracji Michał Boni oraz lider cyfryzacji w Polsce Włodzimierz Marciński. Misją porozumienia miało być działanie na rzecz zdobywania przez obywateli naszego kraju cyfrowych umiejętności niezbędnych zarówno na rynku pracy, jak i w życiu publicznym. Porozumienie zostało objęte honorowym patronatem Prezydenta Bronisława Komorowskiego, a w spotkaniu udział wzięła m.in. wiceprzewodnicząca Komisji Europejskiej i jednocześnie komisarz europejski ds. agendy cyfrowej Neelie Kroes.

Uczestnikom spotkania rozdano arcyciekawy dokument: *Manifest w sprawie e-umiejętności* [67] będący dziełem wielu autorów, do którego wstęp napisał Antonio Tajani, Wiceprzewodniczący Komisji Europejskiej ds. Przemysłu i Przedsiębiorczości. Z tego dokumentu dowiadujemy się, że Europa cierpi na poważny deficyt ludzi z profesjonalnym przygotowaniem informatycznym, a w europejskiej gospodarce cyfrowej marginalizowanych jest 300 milionów osób. Mimo że bezrobocie wśród młodzieży sięga średnio 22%, pracodawcy nie mogą znaleźć pracowników na stanowiska wymagające przygotowania naukowego i technicznego.

W UE tylko niecałe 1,5% osób w wieku 20-29 studiuje nauki ścisłe i techniczne na poziomie akademickim. Wpływ na tę sytuację mają złe programy nauczania, a problem zaczyna się już na poziomie szkoły podstawowej. Informatyka jest w większości krajów unijnych przedmiotem do wyboru. Chlubny wyjątek stanowią Szwajcaria i Austria, gdzie informatyka jest przedmiotem obowiązkowym (w Austrii od pierwszej klasy szkoły podstawowej). A dzieje się tak, mimo że do roku 2015 aż 90% wszystkich stanowisk pracy wymaga podstawowych e-umiejętności (badania International Data Corporation).

Interesujące jest też obalenie mitu o informatycznym przygotowaniu młodych Europejczyków. Okazuje się, że zaledwie 25% młodych ludzi w UE określa siebie jako „mających wysoki poziom kompetencji informatycznych”, przez co rozumieją, że potrafią korzystać z wyszukiwarek internetowych, umieszczać wiadomości na forach dyskusyjnych, wysyłać maile (dumnie podkreślając, że „z załącznikami”), dokonywać zakupu w Internecie plików muzycznych i wymieniać się tymi plikami. W tej sytuacji nie dziwi fakt, że aż 46% do 56% firm we wszystkich sektorach bezskutecznie poszukuje informatyków, którzy zajęliby się ich systemami informatycznymi.

Autorzy *Manifestu* wskazują również na fakt, że kraje takie jak Indie i Chiny, które do tej pory łagodziły deficyt informatyków w Europie (i nie tylko), wkrótce będą potrzebowały tych specjalistów u siebie. A o szybkości z jaką rośnie na świecie zapotrzebowanie na usługi informatyczne może świadczyć choćby fakt, że podczas gdy w roku 2003 wysyłano dziennie 12 milionów maili, dziesięć lat później krążyło ich dziennie już 247 miliardów. Wykładniczo też rośnie wielkość oprogramowania mierzona liczbą linii kodu. Podczas gdy kod Unixa 1.0 w roku

1971 obejmował 20 tys. linii, kod Windows-7 już 40 mln linii, a amerykański portal zdrowotny healthcere.gov — 500 mln (źródło [57]).

Gdzie więc szukać rezerw? Zdaniem autorów *Manifestu* w prawidłowo kształtowanych programach edukacyjnych od szkoły podstawowej po uczelnie wyższe, a także w podniesieniu prestiżu zawodu informatyka na poziomie firm, gdzie informatycy powinni być wykorzystywani w projektach tworzących innowacje, a nie jedynie jako konserwatorzy sprzętu i oprogramowania.

W tym miejscu należy podkreślić, że przy Ministerstwie Edukacji Narodowej od wielu już lat działa Rada ds. Informatyzacji Edukacji, której przewodniczy prof. Jan Madey. Jednym z wyników jej działania jest przygotowanie nowej „Podstawy programowej” dla nauczania informatyki w szkole podstawowej.

15.2 Geniusze i idioci

Od kiedy zacząłem zajmować się informatyką, co nastąpiło z początkiem lat 1960., miałem poczucie, że o informatyce można mówić w sposób zrozumiały. Zawsze też byłem przeciwny takim kwalifikacjom jak „komputerowy geniusz” — na określenie ludzi, którzy nie potrafią mówić o informatyce w sposób jasny, lub też „komputerowy idiota” — odnoszący się do ludzi, którzy komputerowych geniuszy nie rozumieją.

Informatyka nie jest ani trudniejsza, ani łatwiejsza od mechaniki, elektroniki czy chemii, a posługiwanie się będącymi w powszechnym użytku narzędziami informatycznymi w rodzaju edytorów tekstu, arkuszy kalkulacyjnych, systemów obróbki zdjęć czy choćby telefonów komórkowych, nie jest trudniejsze od prowadzenia samochodu, czy żeglowania po Mazurach. Jednakże, podobnie jak w przypadku samochodu i żaglówki, aby się tymi urządzeniami posługiwać, trzeba się czegoś nauczyć. Tymczasem wśród użytkowników „codziennej informatyki” od lat obserwuję dwie sprzeczne ze sobą postawy:

- ja się tego nigdy nie nauczę,
- nie będę czytał podręcznika, sam sobie jakoś poradzę.

Czy ktoś mógłby sobie wyobrazić przyszłego kierowcę, który uważając, że prowadzenia samochodu nigdy nie opanuje, siada do niego i wybiera się na ruchliwą autostradę? Pewnie byłoby to trudne, jednakże dokładniejsza analiza analogicznego zachowania na gruncie informatyki wskazuje, że nie jest ono do końca irracjonalne, gdyż zostało wymuszone przez następujący stan rzeczy:

- podręczniki systemów informatycznych są równie grube jak nieczytelne,
- systemy informatyczne są nam niezbędne do pracy.

Osoby, które wierzą, że świat dzieli się na komputerowych geniuszy i komputerowych idiotów sądzą zapewne, że podręczniki, o których mowa, są niezrozumiałe jedynie dla idiotów. Zapewniam więc te osoby, że mimo iż informatykę zgłębiałem od lat pięćdziesięciu, niewiele tzw. podręczników użytkownika (ang. manuals) byłem w stanie przeczytać ze zrozumieniem. I niezależnie od tego, czy był to podręcznik edytora tekstu, czy zaawansowanego języka programowania, czy opisywał on produkt komercyjny, czy też dostępny w domenie publicznej, najczęściej charakteryzowało go zjawisko, które Anglicy określają w niezbyt elegancki sposób jako „rozwołnienie mowy przy zatwardzeniu myśli”. Mówiąc mniej alegorycznie, był niezwykle rozwlekły, a przy tym bardzo nieprecyzyjny i — co jeszcze dziwniejsze — przy swej rozwlekłości również dalece niekompletny. Aby wyjaśnić to zjawisko w sposób obrazowy posłużę się

analogią z zakresu sztuki kulinarnej. Podręcznik gotowania obejmuje zwykle trzy rodzaje wiedzy:

- wiedzę o surowcach, a więc o składnikach, z których tworzy się potrawy,
- wiedzę o potrawach, a więc o tym, czym one są i jakie powinny mieć cechy,
- wiedzę o narzędziach i sposobach tworzenia potraw z surowców.

Autor podręcznika zakłada też zwykle, że jego czytelnik przeszedł już szkolne kursy botaniki, chemii i fizyki, w związku z czym wie, czym się różni liść od korzenia, co to jest sól i woda i po czym poznać, że woda się gotuje. W swoim podręczniku zawiera więc jedynie tę wiedzę, która wykracza poza obszar podstawowy oraz — co bardzo ważne — pisze o gotowaniu, a nie o „posługiwaniu się kuchnią”, albo — co jeszcze gorzej — o „obsłudze kuchni”¹³⁰.

W odróżnieniu od mistrza kucharskiego, komputerowy geniusz pisze swój podręcznik przy założeniu, że jego czytelnik nie wie nic o składowych produktach, które będzie tworzył, a także nie rozumie do końca jakie to będą produkty. Wie jedynie jak wyglądają one na monitorze komputera. Geniusz komputerowy wychodzi dodatkowo z założenia, że swojego czytelnika (oczywiście komputerowego idioty) i tak żadnych podstaw nie nauczy, wypełnia więc podręcznik listami ruchów, jakie należy wykonać, aby osiągnąć określony obrazek na monitorze. Opisuje sekwencje kliknięć nie troszcząc się o to, by jego czytelnik wiedział, co tak na prawdę robi. Zresztą niejednokrotnie odnoszę wrażenie, że sam geniusz, ma o tym dość blade pojęcie.

Dla przykładu, w większości podręczników języków programowania, które czytałem, każdy element programu nazywany jest „instrukcją”, bez względu na to, czy jest to rzeczywiście instrukcja, czy też deklaracja lub wyrażenie. I dotyczy to nie tylko podręczników takich pospolitaków jak Basic, ale też i takich arystokratów jak prawie już dziś zapomniany Delphi, czy też cieszący się dobrą opinią Python. Kiedyś przekopałem największą akademicką księgarnię w Nowym Jorku w poszukiwaniu napisanego profesjonalnie i dla profesjonalistów podręcznika języka Delphi, i takiego podręcznika nie znalazłem. A były tam podręczniki Delphi liczące po dwa tysiące stron! Jeden z nich kupiłem i srodze się nim rozczarowałem.

Innym razem tworzyłem dla mojej cukierniczej firmy A. Blikle bazodanowy system obsługi logistyki dostaw. Obserwując własne zmagania ze służącym mi do tego narzędziem oceniłem, że około 60% czasu poświęcałem nie na projektowanie architektury systemu, ale na mocowanie się ze składnią języka, która nigdzie nie była jasno opisana. O tym jaka jest dowiadywałem się jedynie z przykładów, które jednak nigdy nie wyjaśniały ogólnej zasady. Podejrzywałem nawet, że być może takiej zasady w ogóle nie było! Bo czym wytłumaczyć fakt, że w jednych instrukcjach parametry oddziela się od siebie przecinkami, a w innych — średnikami. Chyba tylko tym, że jedną instrukcję zaprojektował John, a drugą Bob. O tym, aby składnię języka opisać — jak w latach 1960. — gramatyką bezkontekstową (por. [60]), oczywiście nie mogło być mowy. Podejrzewam też, że ani John ani Bob w ogóle o takich narzędziach nie słyszeli.

Z opisem semantyki (znaczenia) za pomocą przykładów też nie było lepiej. Tu z kolei przypominała mi się anegdota o kimś, kto starał się przekonać swojego rozmówcę, że język niemiecki jest bardzo prosty:

Posłuchaj — mówił — „dzień dobry” to „guten Tag”, „do widzenia” to „auf Wiedersehen”,.... no i tak dalej.

¹³⁰ Od lat staram się wyjaśniać, że „obsługa komputera”, to czynności, które wykonuje serwisant, natomiast użytkownicy komputera posługują się nim, podobnie jak pianista, który nie „obsługuje fortepianu”, ale posługuje się nim gdy gra.

Gdy kupiłem pierwszą wersję edytora Word, spędziłem wiele dni na kilkakrotnym czytaniu podręcznika użytkownika, by wśród powodzi historyjek o klikaniu znaleźć to, co istotne. Znakomicie pomogła mi też znajomość profesjonalnych systemów składu komputerowego TeX i LaTeX, których używałem już od kilku lat. Kiedy odcedziłem ziarno od plew, postanowiłem opisać zdobytą wiedzę w krótkim podręczniku komputerowej edycji dokumentów [26]. Napisałem go nie tylko dla siebie, ale przede wszystkim dla pracowników mojej firmy, którzy — rzecz jasna — nie byli informatykami. Postawiłem sobie wyzwanie napisania podręcznika dla odbiorcy, który co prawda odróżnia myszkę od klawiatury, ale nie ma żadnej głębokiej wiedzy informatycznej.

Podręcznik istnieje do dziś, zawiera około 60 stron tekstu w formacie A4 i obejmuje wszystkie najważniejsze narzędzia Worda służące do edycji dokumentów elektronicznych. A rozpoczyna się od wyjaśnienia czym jest dokument (produkt), czym są tzw. style (główne narzędzia) oraz jak z tych narzędzi korzystać. Ta wiedza, przekazana moim pracownikom, pozwoliła na wprowadzenie jednolitego standardu dla wszystkich tworzonych w firmie dokumentów. Było to możliwe, gdyż dzięki stylom postać graficzna dokumentu (wielkość i krój czcionek, odstępy pionowe i poziome, kształt tabel itp.) jest w dużej mierze automatycznie utrzymywana przez system przy niewielkiej jedynie ingerencji piszącego.

W ciągu ostatnich ponad dwudziestu lat widziałem tysiące nieporadnie pod względem typograficznym napisanych dokumentów i jedynie może kilkanaście napisanych poprawnie. W tych kilkunastu używano stylów w sposób świadomy. W pozostałych, autorzy nie mieli o stylach zielonego pojęcia. Mój podręcznik można bezpłatnie pobrać z witryny <http://www.moznainaczej.com.pl/komputerowa-edycja-dokumentow>.

Oczywiście język podręcznika trzeba dostosować do czytelnika, jego umiejętności i jego potrzeb. Jednym językiem będziemy mówili do kogoś, kto aplikacji komputerowych używa, a innym do informatyka, który je tworzy. Jednego języka użyjemy mówiąc do dorosłych, a innego — mówiąc do dzieci. Za każdym jednak razem, gdy opisujemy jakiś system informatyczny (język programowania lub aplikację), powinniśmy trzymać się porządku pojęciowego, na który składają się trzy kroki:

1. wyjaśnienie czym są **obiekty**, którymi dany system manipuluje (tworzy i przetwarza); np. bazy danych, dokumenty, arkusze kalkulacyjne, pliki video itp.
2. wyjaśnienie jakich **operacji** używamy do tworzenia i przetwarzania obiektów,
3. wyjaśnienie jakim **językiem** komunikujemy się z komputerem, by wskazywać obiekty, które chcemy przetwarzać i operacje, które nam do tego posłużą.

Niestety ogromna większość podręczników systemów informatycznych skupia się na trzecim elemencie, dwa pierwsze pozostawiając domyślności czytelnika. W moim przekonaniu dzieje się tak dlatego, że dokumentacje systemów cierpią na grzech braku precyzji i kompletności zarazem.

Pierwszym krokiem do dobrych podręczników systemów informatycznych pisanych dla użytkowników powinny więc być dobre dokumentacje systemów pisane dla informatyków. Celem, jaki sobie stawiam w niniejszej książce, jest zaproponowanie, a właściwie przypomnienie, matematycznego języka tworzenia takich dokumentacji systemów informatycznych, które mogły być wykorzystane zarówno przez twórców systemów, jak i przez autorów podręczników użytkownika.

16 Bibliografia

- [1] Aalst Wil van der, Hee Kees van, *Workflow management: models, methods, and systems (Cooperative Information Systems)*, MIT Press 2004
- [2] Ahrent Wolfgang, Beckert Bernhard, Bubel Richard, Hähnle Reiner; Schmitt Peter H., Ulbrich Mattias (Eds.), *Deductive Software Verification — The KeY Book; From Theory to Practice*, Lecture Notes in Computer Science 10001, Springer 2016
- [3] Aho A.V., Ullman J.D., *The Theory of Parsing, Translation and Compilation, volume 1, Parsing*, Prentice-Hall, Englewood Cliffs, NJ 1972
- [4] Apt K.R., *Ten Years of Hoare's Logic: A Survey - Part 1*, ACM Trans. Program. Lang. Syst. 3(4): 431-483 (1981)
- [5] Backus J.W., Bauer F.L., Green J., Katz C., McCarthy J., Naur P. (Editor), Perlis A.J., Rutishauser H., Samelson K., Vauquois B., Wegstein J.H., Van Wijngaarden A., Woodger M., *Report on the algorithmic language ALGOL 60*, Numerische Mathematik 2, 106--136 (1960)
- [6] Bakker Jaco (de), *Mathematical Theory of Program Correctness*, Prentice/Hall International 1980
- [7] Banachowski Lech, *Bazy danych. Tworzenie aplikacji*, Akademicka Oficyna Wydawnicza PLJ, Warszawa 1998
- [8] Banachowski Lech, Kreczmar Antoni, Mirkowska Grażyna, Rasiowa Helena, Salwicki Andrzej, *An introduction to Algorithmic Logic — Metamathematical Investigations of Theory of Programs*, T. 2: Banach Centre Publications. Warszawa PWN, 1977, s. 7-99, seria: Banach Center Publications, vol.2
- [9] Bekić Hans, *Definable operations in general algebras and the theory of automata and flowcharts* (manuscript), IBM Laboratory, Vienna 1969
- [10] Binsbergena L. Thomas van, Mosses Peter D., Sculthorped C. Neil, *Executable Component-Based Semantics*, Preprint submitted to JLAMP, accepted 21 December 2018
- [11] Bjørner Dines, Jones Cliff B, *The Vienna development method: The metalanguage*, Prentice Hall International 1982
- [12] Bjørner Dines, Oest O.N. (wydawcy), *Towards a formal description of Ada*, Lecture Notes of Computer Science 98, Springer Verlag 1980
- [13] Blikle Andrzej, *Automaty i gramatyki — wstęp do lingwistyki matematycznej*, PWN 1971
- [14] Blikle Andrzej, *Algorithmically definable functions. A contribution towards the semantics of programming languages*, Dissertationes Mathematicae, LXXXV, PWN, Warszawa 1971
- [15] Blikle Andrzej, *Equational Languages*, Information and Control, vol.21, no 2, 1972

- [16] Blikle Andrzej, *Analysis of programs by algebraic means*, Mathematical Foundations of Computer Science, Banach Center Publications, vol.2, Państwowe Wydawnictwa Naukowe, Warszawa 1977
- [17] Blikle Andrzej, *Toward Mathematical Structured Programming*, Formal Description of Programming Concepts (Proc. IFIP Working Conf. St. Andrews, N.B Canada 1977, E.J Neuhold ed. s. 183-2012, North Holland, Amsterdam 1978
- [18] Blikle Andrzej, *On Correct Program Development*, Proc. 4th Int. Conf. on Software Engineering, 1979 s. 164-173
- [19] Blikle Andrzej, *On the Development of Correct Specified Programs*, IEEE Transactions on Software Engineering, SE-7 1981, s. 519-527
- [20] Blikle Andrzej, *The Clean Termination of Iterative Programs*, Acta Informatica, 16, 1981, s. 199-217.
- [21] Blikle Andrzej, *MetaSoft Primer — Towards a Metalanguage for Applied Denotational Semantics*, Lecture Notes in Computer Science, Springer Verlag 1987
- [22] Blikle Andrzej, *Denotational Engineering or from Denotations to Syntax*, red. D. Bjørner, Jones Cliff B, M. Mac an Airchinnigh, E.J. Neuhold, *VDM: A Formal Method at Work*, Lecture notes in Computer Science 252, Springer, Berlin 1987
- [23] Blikle Andrzej, *Three-valued Predicates for Software Specification and Validation*, w tomie VDM'88, VDM: The Way Ahead, Proc. 2nd, VDM-Europe Symposium, Dublin 1988, Lecture Notes of Computer Science, Springer Verlag 1988, s. 243-266
- [24] Blikle Andrzej, *Denotational Engineering*, Science of Computer Programming 12 (1989), North Holland
- [25] Blikle Andrzej, *Why Denotational — Remarks on Applied Denotational Semantics*, Fundamenta Informaticae 28, 1996, s. 55-85
- [26] Blikle Andrzej, *Komputerowa edycja dokumentów dla inteligentnych*, Podręcznik napisany dla pracowników firmy A. Blikle, Wersja 1.2 dla Word 2007; dokument dostępny na witrynie autora pod adresem <http://moznainaczej.com.pl/komputerowa-edycja-dokumentow>
- [27] Blikle Andrzej, *Doktryna jakości*, Helion One Press, wyd. I w roku 2014 i wyd. II w roku 2017; wersja cyfrowa dostępna na witrynie autora <http://www.moznainaczej.com.pl/wydanie-drugie-dj/co-w-drugim-wydaniu>
- [28] Blikle Andrzej, Mazurkiewicz Antoni, *An algebraic approach to the theory of programs, algorithms, languages and recursiveness*, Proc. International Symposium and Summer School on Mathematical Foundations of Computer Science, Warsaw-Jablonna, 1972.
- [29] Blikle Andrzej, Tarlecki Andrzej, *Naive denotational semantics*, Information Processing 83, R.E.A. Mason (ed.), Elsevier Science Publishers B.V. (North-Holland), © IFIP 1983
- [30] Blikle Andrzej, Tarlecki Andrzej, Thorup Mikkel, *On conservative extensions of syntax in system development*, Theoretical Computer Science 90 (1991), 209-233
- [31] Branquart Paul, Luis Geoges, Wodon Pierre, *An Analitical Description of CHILL, the CCITT High Level Language*, Lecture Notes in Computer Science vol. 128, Springer-Verlag 1982

- [32] Chomsky Noam, *Three models for the description of language*, IRE Transactions of Information Theory, IT2, 1956
- [33] Chomsky Noam, *Syntactic Structures*, Hague 1957
- [34] Chomsky Noam, *On certain formal properties of grammars*, Information and Control, 2, 1959
- [35] Chomsky Noam, *Context-free grammar and pushdown storage*, MIT Research Laboratory Electrical Quarterly Progress Reports 65, 1962
- [36] Cohn Paul M., *Uniwersal Algebra*, D. Reidel Publishing Company 1981
- [37] Dijkstra Edsger, W., *goto statements considered harmful*, Communications of ACM, 11, 1968, s. 147-148
- [38] Dijkstra Edsger, W., *A constructive approach to the problem of program correctness*, BIT 8 (1968)
- [39] DuBois Paul, *MySQL*, Wydanie II rozszerzone, Mikom, Warszawa 2004
- [40] Floyd Richard W., *Assigning meanings to programs*, Appl. Math. Comput. 19, 1967, s. 19-32
- [41] Forta Ben, *SQL w mgnieniu oka*, Helion 2015
- [42] Ginsburg Seymour, *The mathematical theory of context-free languages*, New York 1966
- [43] Ginsburg Seymour, Rice, H.G., *Two Families of Languages Related to Algol*, Journal of the Association of Computing Machinery, 9 (1962)
- [44] Goguen, J.A., Thatcher J.W., Wagner E.G., Wright J.B., *Initial algebra semantics and continuous algebras*, Journal of ACM 24 (1977)
- [45] Gordon M.J.C., *The Denotational Description of Programming Languages*, Springer Verlag, Berlin 1979
- [46] Gruber Martin, *SQL*, Helion 1996
- [47] Hoare C.A.R., *An axiomatic basis for computer programming*, Communications of ACM, 12, 1969, s. 576-583
- [48] Jensen Kathleen, Wirth Niklaus, *Pascal — User Manual and Report*, Springer Verlag 1975
- [49] Kleene Steven Cole, *Introduction to Metamathematics*, North Holland 1952; później wznawiane w latach 1957, 59, 62, 64, 67, 71
- [50] Konikowska Beata, Tarlecki Andrzej, Blikle Andrzej, *A tree-valued Logic for Software Specification and Validation*, w tomie VDM'88, VDM: The Way Ahead, Proc. 2nd, VDM-Europe Symposium, Dublin 1988, Lecture Notes of Computer Science, Springer Verlag 1988, s. 218-242
- [51] Landin, P. *The mechanical evaluation of expressions*, BSC Computer Journal, 6 (1964), 308-320
- [52] Leszczyłowski Jacek, *A theorem of resolving equations in the space of languages*, Bull. Acad. Polonaise de Science, Série de Sci. Math. Astronom. Phys. 19 (1971).
- [53] Madey Jan, *Od wnioskowania gramatycznego do walidacji specyfikacji wymagań*, w tomie „Symulacja w badaniach i rozwoju”, tom 6, Politechnika Białostocka; na

Researchgate https://www.researchgate.net/publication/283225534_Od_wnioskowania_gramatycznego_do_validacji_specyfikacji_wymagan_From_grammatical_inference_to_validation_of_requirements_specification

- [54] Madey J., Matwin S., *Pascal — opis języka*, Sprawozdania IInf UW nr 54 oraz 55, Wydawnictwa Uniwersytetu Warszawskiego, Warszawa 1976
- [55] Mazurkiewicz Antoni, *Proving algorithms by tail functions*, Information and Control, 18, 1971, s. 220-226
- [56] McCarthy John, *A basis for a mathematical theory of computation*, Western Joint Computer Conference, May 1961 później opublikowane w Computer Programming and Formal Systems (pod redakcją P. Brawffort i D. Hirschberg), North Holland 1967
- [57] Microsoft Press (opr. w. polskiej Piotr Stokłosa), *Microsoft Access 2000 — wersja polska*, Wydawnictwo RM, 2000
- [58] Niemiec Andrzej, *Wielkość współczesnego oprogramowania*, Biuletyn PTI nr 4-5, 2014
- [59] Norton Peter, Samuel Alex, Aitel David, Eriv Foster-Johnson, Richardson Leonard, Diamond Jason, Parker Aleatha, Michael Roberts, *Pyton od podstaw*, Wydawnictwo Helion 2006
- [60] Parnas D.L., Asmis G.J.K., Madey J., *Assessment of Safety-Critical Software in Nuclear Power Plants*, Nuclear Safety 32, 2, April-June 1991, str. 189-198.
- [61] Paszkowski Stefan, *Język ALGOL 60*, PWN 1965
- [62] Sephens Ryan, Jones D. Arie, Plew Ron, *SQL w 24 godziny*. Helion 2016
- [63] Stoy, Joseph E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA 1977
- [64] Scott D., Strachey Ch., *Towards a mathematical semantics of computer languages*, Technical Monograph PRG-6, Oxford University 1971.
- [65] Tarski Alfred, *Pojęcie prawdy w językach nauk dedukcyjnych*, Prace Towarzystwa Naukowego Warszawskiego, Nr 34, Wydział III, 1933, str.35
- [66] Turing Alan, *On checking a large routine*, Report of a Conference on High Speed Calculating Machines, University Mathematical Laboratory, Cambridge 1949, s. 67-69.
- [67] Vera (del) Pilar Castillo, Curley Martin, Fabry Eva, Gottiz Michael, Hagedorn Peter, Herczog Edit, Higgins John, Joyce Alexa, Korte, Werner, Lanvin Bruno, Parola Andrea, Straub Richard, Tapscott Don, Vassallo John, *Manifest w sprawie e-umiejętności*, European Schoolnet (EUN Partnership AISBL)
- [68] Viescas John, *Podręcznik Microsoft Access 2000*, wydawnictwo RM 2000

17 Skorowidze

17.1 Skorowidz pojęć i nazwisk

Ada	110	aktualizacja funkcji.....	36
Aho A.V	68, 358	aktualne parametry referencyjne...	189, 190

aktualne parametry wartościowe ..189, 190, 196	definicja dopuszczalna w preambule 234
alfabet42, 44	definicja semantyki algebraiczna..... 163
algebra abstrakcyjna56	definicja semantyki wyraźna 163
algebra bezkontekstowa69	deklaracja dopuszczalna w preambule.. 234
algebra danych.....116	deklaracja obiektu 267
algebra denotacji wyrażeń146	deklaracja procedury funkcyjnej..... 208
algebra jednoznaczna66	denotacja 99
algebra kompozytów124	denotacja aplikatywna 141
algebra korpusów118	denotacja imperatywna 141
algebra mniej lub tak samo wieloznaczna67	Dijkstra Edsger W..... 76, 79, 360
algebra osiągalna65	drzewo parsingu..... 161
algebra pusta.....60	DuBois Paul 274, 360
algebra składniowa69	diedzina prosta 113
algebra struktur danych55	diedzina strukturalna 113
algebra typów135	diedziny zwrotne 101
algebra wielorodzajowa.....58	element dolny zbioru 40
algebra wieloznaczna66	element maksymalny 39
algebraiczny model struktur danych111	element minimalny 39
Apt Krzysztof2, 240, 358	element najmniejszy 39
arność symbolu funkcji58	element największy..... 39
Arystoteles.....52	epimorfizm..... 61
asercja.....88, 230	ewaluacja leniwa..... 53
atrybut rekordu113	ewaluacja ochocza 53
Bakker Jaco de240, 358	Floyd Richard 76, 263, 360
Banachowski Lech274, 287, 290, 358	formalne parametry referencyjne..... 190
Bekić Hans41	formalne parametry wartościowe 190
Bekić Hansa.....41	funkcja całkowita..... 32
Bekić'a-Leszczylowskiego twierdzenie ..42	funkcja ciągła..... 40
biblioteka obiektów265	funkcja częściowa..... 32
błąd abstrakcyjny.....51	funkcja monotoniczna..... 40
całkowity postwarunek.....83	funkcja odwracalna 47
całkowity prewarunek83	funkcja skończona 32
Chomsky Noham.....42, 45, 360	funkcja szkieletowa 68
ciało procedury191	funkcja wzajemnie jednoznaczna 47
Cohn Paul M.....50, 360	funkcji klanowania korpusów..... 118
Curry Haskell33	Ginsburg Seymour 42, 360
czas deklaracji procedury191	Goguen Joe 62, 360
czas wywołania procedury191	Gordon Michael 102, 191, 240
częściowy postwarunek.....83	górne ograniczenie zbioru..... 39
częściowy prewarunek83	graf podrzędności..... 297
Dahl Ole-Johan.....256	gramatyka bezkontekstowa..... 42
dana dobrze ustrukturyzowana.....123	gramatyka równaniowa..... 44, 45
dana dobrze utypowiona.....139	granica łańcucha 39
dana prosta.....113	graniczeniem górnym zbioru 39
dana pusta.....294	hipoteza Collatza 85
dana strukturalna113	Hoare.....28
dana ustrukturyzowana.....123	Hoare C.A.R. 1–357
dana utypowiona139	homomorfizm konstytuujący 64
	homomorfizm szkieletowy 71
	homomorfizm wielorodzajowy..... 60

identyfikator rejestrowy	257	metaprogram poprawny	238
indukcja strukturalna	99	MetaSoft	16, 30, 103
inicjał korpusu	117	metawarunek	223, 234
instrukcja atomowa	230	metoda pięciu kroków	107
instrukcja skoku	78, 100	mocny niezmiennik	235
instrukcja specyfikowana	229	model komutuje	105
instrukcja trywialna	172	monomorfizm	61
interpreter	162	Mosses Peter	102
izomorfizm	61	MPP	239
jarzmo	127	nadalgebra algebry	60
jądro homomorfizmu	61	najsilniejszy postwarunek częściowy	83
język atomowy	45	najsłabszy prewarunek całkowity	84
języki równaniowo definiowalne	45	nałożenie	61
kartezjańska potęga zbioru	32	niezmiennik pętli	97
klan korpusu	118	niezmiennik rejestrowy	257
klan typu	135	nośnik algebry	55, 59
Kleene Stephen Cole	40	notacja infiksowa	63
Kleene'ego twierdzenie	40	notacja prefiksowa	63
Kleene Steven Cole	14, 89, 225, 277	obiekt	265
koherentności relacja	169	obszar obowiązywania warunku	231
kolokwializm	107	obszar widoczności zmiennych	189
kompilator	162	odwołanie do biblioteki	267
kompozycyjność semantyki	99	odwrotność relacji	47
kompozyt	123	ograniczenie funkcji	33
konkatenacja	43	operacja teoretyczna	113
konkretyzacja	107	operacja transparentna	52
konstruktory programistyczne proste	86	parametrów wołanie	186
konstruktory programistyczne		parser	161
rekurencyjne	86	Plotkin Gordon	100
konstruktory strukturalne	79	podalgebra	60
kontekst logiczny	237	podalgebra osiągalna	64
kontynuacje	100	podobne algebry	60, 62
korpus	117	podobne sygnatury	61
korpus wewnętrzny	117	podprogram	186
kres górny zbioru	39	<i>poprawność całkowita programu</i>	83
krotka	36	<i>poprawność częściowa programu</i>	83
kwantyfikatory egzystencjalny	31	poprawność programu	76
kwantyfikatory ogólny	31	porządek dobrze ufundowany	39
Landin Peter	62, 360	potęg języka	43
Leszczyłowski Jacek	41, 360	preambuła poprawna	234
łańcuch	39	preambuły rozdzielne	234
Madey Jan	2, 355, 360, 361	procedura	189
mapping	32	procedura funkcyjna	186, 190, 208
Mazurkiewicz Antoni	2, 76, 77, 100, 344, 359, 361	procedura imperatywna	186, 190
McCarthy John	14, 20, 21, 53, 54, 55, 62, 80, 126, 130, 131, 153, 223, 224, 225, 235, 277, 361	procedura rekurencyjna	81, 186
metapredykat	234	program iteracyjny	78, 79
metaprogram	223, 238	program prefiksowany	269
		program strukturalny	76
		programowanie strukturalne	101, 187
		przechodność	38

przesłonięcie funkcji	36	słaba antysymetria	38
przywołanie obiektu	268	słaba spełnialność predykatu	236
pseudodana	139	słaby niezmiennik	235
pseudokompozyt.....	139	słowo nad alfabetem	42
pseudowartość	139	Sokołowski Stefan	263
quasiporządek.....	38	specinstrukcja	229
reguła kopiowania (ang. copy rule).....	101	stała typologiczna	111, 169
rejestr	257	stan	139
rekord korpusowy.....	117	stan adekwatny.....	228
rekurencja prosta	188	stan adekwatny wzgl. preambuły.....	228
rekurencja wzajemna	188	stan inicjalny	228
rekurencją prosta	81	Stoy Joseph E.....	101
rekurencyjne wywołanie procedury	188	Strachey Christopher	101, 102
relacja antysymetryczna	97	sygnatura algebry.....	58
relacja binarna	46	syntaksa języka	99
relacja identycznościowa.....	46	szkielet funkcji.....	68
relacja podrzędność tabel	296	środowisko	140
relacja przeciwwrotna.....	97	tabela.....	294
relacja pusta.....	46	tabela nadrzędna	296
relacja wejście-wyjście.....	78	tabela podrzędna	296
rodzaj danych	113	tabela pusta	295
rodzaj symbolu funkcji.....	58	tabelowy konstruktor kolumnowy	301
rozszerzenie algebry.....	59	tabelowy konstruktor wierszowy.....	301
rozszerzenie sygnatury	59	Tarlecki Andrzej 2, 90, 102, 123, 176, 191, 199, 207, 238, 359, 360	
równania lewostronnie liniowe	78	termy	102
rzetelna denotacja imperatywna	170	teza.....	223, 233
rzetelny konstruktor denotacji wyrażeń	141	Thacher Jim	62
SAKO	186	transakcja	328
SAKO - język programowania	186	transfer	127
SAKO - System Automatycznego Kodowania	186	transformacja przywracająca	107
Scott Dana	101, 102, 103, 361	treść procedury.....	189
semantyka.....	107	Turing Alan.....	12, 76, 85
semantyka abstrakcyjna.....	107	typ	134
semantyka dynamiczna	110	typ bezjarzmowy.....	135
semantyka konkretna.....	107	typ zadeklarowanej zmiennej	168
semantyka operacyjna	100	Ullman J. D.....	68
semantyka statyczna	110	VDL	99
sieć działań	78	VDM	99
silna spełnialność predykatu.....	236	Wagner Eric	62, 360
skład	140	warstwa deskryptywna.....	222
składni abstrakcyjna	62	warstwa programistyczna	222
składnia abstrakcyjna algebry	63	wartościowanie	140
składnia języka	99	wartościowanie zmiennych.....	101
składnia kolokwialna.....	107	wartość	139
składnia konkretna.....	66, 107, 152	wartość właściwa	139
składniowa kategoria wyrażeń	62	warunek.....	223
składowe homomorfizmu	60	warunek algorytmiczny	229
składowe procedury funkcyjnej.....	208	warunek danologiczny	224
składowe procedury imperatywnej.....	200	warunek rejestrowy.....	257

warunek stopu pętli	96	wyrażenie typologiczne	111
warunek w postaci standardowej.....	229	wywołanie procedury.....	189
warunek walidacyjny.....	226	wywołanie procedury funkcyjnej	208
wieloprocedura	204	<i>zasada prostoty języka</i>	104
wielorodzajowy język formalny	44	zbiór częściowo uporządkowany	38
własność stopu programu	84	zbiór dobrze uporządkowany.....	39
właściwość	223	zbiór łańcuchowo ograniczony	97
właściwość składniowa	233	zbiór łańcuchowo zupełny	40
włożenie	61	złożenie sekwencyjne relacji	46
Wright Jessie	62, 360	ZŁZ (zbiór łańcuchowo zupełny)	40
wyrażenie danologiczne	111, 141	zmienna.....	142, 168
wyrażenie eksportujące	207	zmienna danologiczna	111, 168
wyrażenie obiektowe	266	zmienna typologiczna	145
wyrażenie rejestrowe	257	zwrotność	38

17.2 Skorowidz oznaczeń

ε : słowo puste	$\emptyset$: zb. pusty i rel. pusta	$\blacklozenge$: przesłonięcie funkcji
$\subset$: być podzbiorem	$\sqsubseteq$: częściowy porządek	@ : formuła algorytmiczna
$\rightarrow$: funkcje częściowe	Θ : element pusty	■ : koniec twierdzenia
$\mapsto$: funkcje całkowite	$\{a.i \mid i=1;n\}$: zbiór	
$\Rightarrow$: funkcje skończone	$(a.i \mid i=1;n)$: ciąg	
• : złożenie relacji	$[a.i/b.i \mid i=1;n]$: mapping	
© : konkatenacja krotek	$Rel.(A,B)$: zb. relacji	
$\exists$: istnieje	$[A]$: podzbiór rel. identyczności	
$\forall$: dla każdego		

17.3 Skorowidz dziedzin i algebr

Ten skorowidz służy głównie autorom książki dla zachowania konsekwencji w oznaczaniu dziedzin i ich elementów.

Algebra danych, rozdział 5.2.1

AlgDan

boo : Boolowska

lic : Liczba

sło : Słowo

lis : Lista

tab : Tablica

rek : Rekord

dan : Dana

ide : Identyfikator

Algebra korpusów, rozdział 5.2.2

AlgKor

kor : Korpus

rodzaj : KorpusB $\mapsto \{('boolowska'), ('liczba'), ('słowo'), 'L', 'T', 'R'\}$

Kr[o]pe : Korlde-1 x ... x Korlde-n $\mapsto$ KorpusB

Algebra kompozytów, rozdział 5.2.3

AlgKom

kom : KompozytB

kom : KompozytBoo

rodzaj.(dan, kor) = rodzaj.kor

Km[o]pe : Komlde-1 x ... x Komlde-n $\mapsto$ KompozytB

Algebra transferów, rozdział 5.2.4

AlgKom

jar : Jarzmo

tra : Transfer

Kt[o]pe.(tra-1, ..., tra-n).kom = Km[o]pe.(tra-1.kom, ..., tra-n.kom)

Algebra typów, rozdział 5.2.5

AlgTyp

typ : Typ

Ky[...]

Wartości i stany pamięci, rozdział 5.3.1

war : Wartość

sta : Stan

śro : Środ

śrt : ŚroTyp

śrp : ŚroPro

skł : Skład

wrt : Wart

Denotacje wyrażeń danologicznych, rozdział 5.3.2

dwd : DenWyrDan

Denotacje wyrażeń typologicznych, rozdział 5.3.4

dwt : DenWyrTyp

Algebra denotacji wyrażeń danologicznych i typologicznych, rozdział 5.3.5

AlgDenWyr

tra : DenWyrTra = Transfer

Składnia abstrakcyjna języka Lingua-A, rozdział 5.4.1

AlgWyrA — algebra wyrażeń składni abstrakcyjnej

ide : Identyfikator

wyd : WyrDanA

wtr : WyrTraA

wyt : WyrTypA

Składnia konkretna języka Lingua-A, rozdział 5.4.2

AlgWyr — algebra wyrażeń składni konkretnej

ide : Identyfikator

wyd : WyrDan

wtr : WyrTra

wyt : WyrTyp

Szkic semantyki Lingua-A, rozdział 5.7

AlgDenWyr — algebra denotacji wyrażeń

Sid : Identyfikator $\mapsto$ Identyfikator

Swd : WyrDan $\mapsto$ DenWyrDan

Swtr : WyrTra $\mapsto$ DenWyrTra

Swt : WyrTyp $\mapsto$ DenWyrTyp

Dziedziny denotacyjne Lingua-1, rozdział 6.1.1

ddz : DenDekZmi

ddt : DenDefTyp

din : DenIns

dpe : DenPre

dpr : DenPro

Składnia abstrakcyjna Lingua-1, rozdział 6.2.1

dez : DekZmiA

det : DefTypA

ins : InstrukcjaA

pab : PreambułaA

prg : ProgramA

Składnia konkretna Lingua-1, rozdział 6.2.2

dez : DekZmi

det : DefTyp

ins : Instrukcja

pab : Preambuła

prg : Program

Semantyka, rozdział 6.3

Sdz : DekZmi $\mapsto$ DenDekZmi

Sdt : DefTyp $\mapsto$ DenDefTyp

Sin : Instrukcja $\mapsto$ DenIns

Spre : Preambuła $\mapsto$ DenPre

Spr : Program $\mapsto$ DenPro

Procedury, rozdział 7.1.4

pfo : ParFor

pak : ParAkt

pri : ProImp

prf : ProFun

pro : Procedura

ddp : DenDekPri

ddf : DenDekPrf

Procedury funkcyjne 7.5.2

ddf : DenDekPrf = Stan $\mapsto$ Stan (denotacje deklaracji procedur funkcyjnych)

prf : ProFun = ParAkt $\mapsto$ Stan $\rightarrow$ KompozytB (procedury funkcyjne)

spf : SkładowePrf = Identyfikator x ParFor x DenPro x DenWyrDan x DenWyrTyp

Warunki, rozdział 8.2

wrn : Warunek

Swa : Warunek $\mapsto$ Stan $\rightarrow$ Kompozyt | Błąd

sin : SpecInstrukcja

Ssi : SpecInstrukcja $\mapsto$ Stan $\rightarrow$ Stan

mep : MetaProgram

Smp : MetaProgram $\mapsto$ {pp, ff}

Obiekty, rozdziały od 9.1 do 9.5

obi : Obiekt = Stan $\mapsto$ Stan

bio : BibObi = Identyfikator $\Rightarrow$ Obiekt

dwo : DenWyrObi = BibObi $\mapsto$ Obiekt | Błąd

ddo : DenDekObi = BibObi $\mapsto$ BibObi | Błąd

dpo : DenPrzObi = BibObi $\mapsto$ Obiekt

dpp : DenProPre = BibObi x Stan $\rightarrow$ Stan

Składnia związana z obiektami, rozdział 9.6

wyo : WyrObi

deo : DefObi

pob : Przywołanie

prx : ProgramPrefiksowany

Dane, korpusy i kompozyty Lingua-SQL, rozdział 12.2.1

dat : Data

cza : Czas

dcz : DataCzas

rok : Rok

mie : Miesiąc
dzi : Dzień
god : Godzina
min : Minuta
sek : Sekunda

dap : DanProsta
wie : Wiersz
tab : Tabela

kor : KorProsty
kor : KorWiersz
kor : KorTabela
kom : KomProsty

Relacje podrzędności w Lingua-SQL, rozdział 12.2.2

grp : GrafPod

Wartości w Lingua-SQL, rozdział 12.6

rbd : RekBazDan

Nowe dziedziny składni konkretnej Lingua-SQL, rozdział 12.8

trn : Transakcja
que : Query